Indian Institute of Technology, Indore
Department of Computer Science and Engineering.

CS-254 Lab Project

Cache Oblivious Algorithms

Tarun Gupta (180001059)
Kartik Garg (180002027)

03-06-2020

Contents

1. Introduction

1.1 Background

A cache-oblivious algorithm (or cache-transcendent algorithm) is an algorithm designed to take advantage of a CPU cache without having the size of the cache (or the length of the cache lines, etc.) as an explicit parameter [5]. As shown in Fig. 1.1, B represents the size of a block or a cache line and M represents the total cache size. Therefore, a cache of size M has $\frac{M}{B}$ blocks.

To compute something, data required for that computation must be transferred to cache memory from Disk. Transfers happen in blocks of B elements. It's not possible to divide a block in further into more pieces. The cache blocks are analogous to a 'quantum' in quantum mechanical theory.

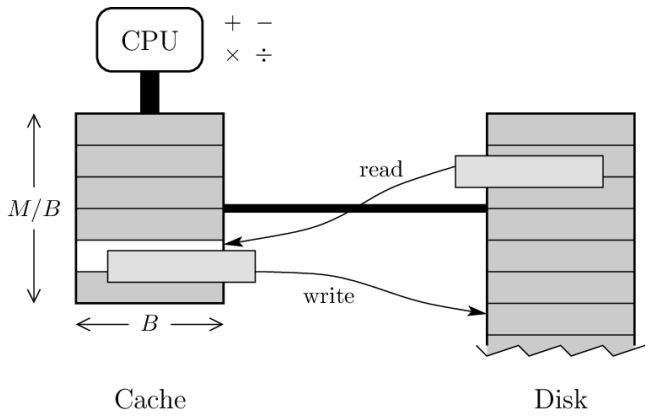

Figure 1.1: Two-level cache model shown with one block replacing another [4]

1.2 Motivation

In traditional algorithmic design classes, we never take into consideration memory transfers while designing an algorithm. As a result, most of the algorithms perform very poorly with

respect to memory transfer complexity. However, memory transfers are very expensive and therefore must be dealt with rigour as we deal with time complexity.

We could explicitly write and manage our data in different levels of cache. These are known as Cache aware algorithms. But they have their own set of problems:

- Cache aware algorithms require separate software other than a normal programming language and lot of expertise of deal with.
- Transfer-ability of code decreases. Code needs to be customised for each system.

Therefore, there is a need to create algorithms which are as cache-efficient as cache-aware algorithms, while still being easy to code and easy to transfer from one system to another.

1.3 History

Charles E. Leiserson of MIT was the first to think about cache-oblivious algorithms in 1996. And first paper in this topic was field was published by Harald Prokop in 1999 in his master's thesis. In this paper, he analyzed and discussed some problems in detail.

1.4 Literature review

Seeing the importance of cache oblivious algorithms, various research articles has been published in this domain. Some of which are briefly discussed below:

- Cache oblivious algorithm's pioneering paper was published in ACM Transactions on Algorithms [5] by Frigo et al. describing various cache oblivious algorithms such as matrix multiplication, matrix transpose, funnel-sort, Van Emde Boas static search tree, etc.
- Erik D. Demaine described several of the results of cache oblivious algorithms along with the intuition behind their design in his study material [4], which we extensively use. Some of the figures and proofs are directly taken form here, we have cited it where-ever needed.
- Brodal et al. worked on cache oblivious sorting [3]. They investigated Lazy funnel-sort algorithm emperically by conducting various experiments. He also compared funnelsort with some recent cache-aware algorithms to see it's efficacy.
- Blelloch et al. [2] explored the area of cache oblivious algorithms for parallel computing.

- Bader et al. [1] worked on finding cache oblivious matrix multiplication. They investigated use of Peano Indexing for matrix multiplications. Some of the figures and code for Peano multiplication is directly taken from here and cited appropriately.

2. Cache Replacement Policy

When the cache is full, there must be a policy to choose which blocks to discard to make room for the new ones. There are various cache replacement policies such as First In First Out (FIFO), Least Frequently Used (LFU), Least Recently Used (LRU) etc. Nowadays, most of the modern computers use LRU with slight varieties such as Pseudo-LRU (PLRU) etc. Competitiveness of LRU was originally proved Daniel D. Sleator and Robert E. Tarjan [7]. We provide a similar proof of LRU below:

Theorem 1 *If an algorithm which uses optimal replacement strategy with cache size of $M/2$ requires T memory transfers, then it takes LRU atmost $2T$ memory transfers with a cache size of M (with same block size B), i.e. :*

$$LRU_M \leq 2 \cdot OPT_{M/2}$$

here LRU_M refers to number of memory transfers with LRU cache replacement policy, with cache size M. $OPT_{M/2}$ refers to number of memory transfers with optimal cache replacement policy with cache size $M/2$.

Proof:

Divide the timeline of accesses of blocks into maximal phases of M/B distinct block accesses.

For any phase ϕ, $LRU_M(\phi) \leq M/B$. This upper bound M/B is reached when there are no common blocks between phase ϕ and the one preceding it. In case, there are common blocks between phase ϕ and the one preceding it, the number of memory transfers may be less than M/B.

For any phase ϕ, $OPT_{M/2} \geq \frac{M}{2B}$. This lower bound $\frac{M}{2B}$ occurs when all the blocks in cache from phase preceding ϕ is required at the start of the phase ϕ. Since, the size of cache is $M/2$ and not M, therefore the first $\frac{M}{2B}$ accesses are free (already in cache from previous phase), and the next $\frac{M}{2B}$ have to be loaded in the cache.

Therefore, $LRU_M \leq 2 \cdot OPT_{M/2}$.

3. Valgrind software for cache-efficiency analysis

Valgrind [6] is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.

Cachegrind is a tool provided in Valgrind, that simulates how your program interacts with a machine's cache hierarchy. It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines [6].

For modern machines which have three or four levels of cache, Cachegrind simulates the first-level and last-level caches. The reason for this choice is that the last-level cache has the most influence on runtime, as it masks accesses to main memory. Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches [6].

Cachegrind gathers the following statistics:

- I1: instruction cache statistics for L1 cache.
- LLi: instruction cache statistics for Last-level cache.
- D1: data cache statistics for L1 cache.
- LLd: data cache statistics for Last-level cache.
- LL: combined (instruction + data) cache statistics for L1 cache.

Instruction caches contain program instructions, meaning assembly instructions, whereas data caches contain data.

For illustrating the working of Valgrind software, we have shown a sample output of running Valgrind on a simple matrix multiplication program written in c++ language in Fig. 3.1.

```
Dimension: 729
Time taken: 6.49588s
GARGs-MacBook-Air:Non-Cache-Oblivious garg$ valgrind --tool=cachegrind ./nor
malMatrixMult 10000000
==3549== Cachegrind, a cache and branch-prediction profiler
==3549== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et a
l.
==3549== Using Valgrind-3.16.0.GIT and LibVEX; rerun with -h for copyright i
nfo
==3549== Command: ./normalMatrixMult 10000000
==3549==
--3549-- warning: L3 cache found, using its data for the LL simulation.
Dimension: 729
Time taken: 271.61s
==3549==
==3549== I   refs:      48,049,756,090
==3549== I1  misses:             6,892
==3549== LLi misses:             4,271
==3549== I1  miss rate:           0.00%
==3549== LLi miss rate:           0.00%
==3549==
==3549== D   refs:      28,628,608,407  (18,831,853,688 rd   + 9,796,754,719
 wr)
==3549== D1  misses:       409,618,082  (   409,446,442 rd   +       171,640
 wr)
==3549== LLd misses:           280,491  (       109,799 rd   +       170,692
 wr)
==3549== D1  miss rate:            1.4% (           2.2%     +           0.0
%  )
==3549== LLd miss rate:            0.0% (           0.0%     +           0.0
%  )
==3549==
==3549== LL refs:          409,624,974  (   409,453,334 rd   +       171,640
 wr)
==3549== LL misses:            284,762  (       114,070 rd   +       170,692
 wr)
==3549== LL miss rate:             0.0% (           0.0%     +           0.0
%  )
```

Figure 3.1: Sample output of Valgrind after running it on a matrix multiplication c++ program.

4. Experiments and analysis

In this section, we conduct experiments and critical analysis of the results obtained for various algorithms.

4.1 Median Selection

4.1.1 Algortihm

For finding median of an unsorted array, Cache-Oblivious Algorithm is quite similar as the traditional one with $O(N)$ time complexity, with difference in implementation. It requires $O(1 + \frac{N}{B})$ memory transfers. The pseudo-code for this algorithm is shown here.

Algorithm 1: Algorithm for picking k^{th} element.

1: Pick an index in the array. The element at this index is called the pivot.
2: Split the list into 2 groups:
   a: Elements less than or equal to the pivot, lows.
   b: Elements strictly greater than the pivot, highs.
3: We know that one of these groups contains the median. Suppose we're looking for the k^{th} element:
   a: If there are k or more elements in lows, recurse on array lows, searching for the k^{th} element.
   b: If there are fewer than k elements in lows, recurse on array highs. Instead of searching for k, we search for k-length(lows).

Algorithm 2: Algorithm for picking pivot.

1: Divide array in smaller sub-arrays of size 5.
2: For each these sub-arrays, calculate their medians and store them in medians.
3: Calculate median of medians and return it.

Algorithm 3: Algorithm for median selection.

1: If length(array) is odd, select $(\frac{length(array)}{2})^{\text{th}}$ element from array.
2: Else, select $(\frac{length(array)}{2})^{\text{th}}$ and $(\frac{length(array)}{2} + 1)^{\text{th}}$ element from array and average them.

4.1.2 Memory Transfer Complexity

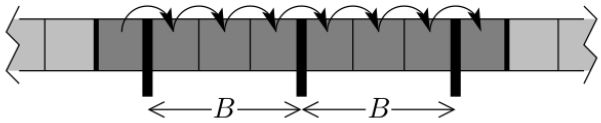

Figure 4.1: Scanning an array of N elements arbitrarily aligned with blocks of size B [4]

Theorem 2 *Scanning an array of size N requires $O(N/B + 1)$ memory transfers [4].*

Proof: As shown in the Fig. 4.1, the array elements are divided in blocks of size B. In the worst case, we will have 2 unfilled blocks at the start and at the end, which requires $\lceil N/B \rceil + 1$ blocks.

Theorem 3 *Memory transfer complexity of finding median in an unsorted array is $O(1 + \frac{N}{B})$ [4].*

Proof: In median selection, for each sub-problem of size N, for picking the pivot, we have to scan our array of size N and write smaller sub-arrays. Memory transfer complexity for scanning is $O(1 + \frac{N}{B})$. Then we have to find median of an array of medians which is basically a sub-problem of size $\lceil \frac{N}{5} \rceil$. After picking pivot, we have to scan our array of size N and split it into two arrays of lows and highs. Memory transfer complexity of scanning is again $O(1 + \frac{N}{B})$. Also the maximum size of lows or highs can be $\lceil \frac{7N}{10} \rceil$.
Hence we obtain a recurrence on memory transfers for an problem of size N as:

$$MT(N) = MT(\frac{N}{5}) + MT(\frac{7N}{10}) + O(1 + \frac{N}{B})$$

We use the base case: $MT(O(B)) = O(1)$. This step is $O(1)$ because now the all the required data fits inside the cache and therefore there are no more memory transfers required. On opening the recursion tree and summing all the levels, we get $MT(N) = O(1 + \frac{N}{B})$.

4.1.3 Time Complexity Analysis

This time complexity is analysis is very similar to the Memory Complexity Analysis.

In median selection, for each sub-problem of size N, for picking the pivot, we have to scan our array of size N and write smaller sub-arrays. Memory transfer complexity for scanning is N. Then we have to find median of an array of medians which is basically a sub-problem of size

$\lceil \frac{N}{5} \rceil$. After picking pivot, we have to scan our array of size N and split it into two arrays of lows and highs. Memory transfer complexity of scanning is again N. Also the maximum size of lows or highs can be $\lceil \frac{7N}{10} \rceil$.

Hence we obtain a recurrence on memory transfers for an problem of size N as:

$$T(N) = T(\frac{N}{5}) + T(\frac{7N}{10}) + N$$

We will add an extra condition here:
For $N \leq 140$:

$$T(N) = O(1)$$

Else:

$$T(N) = T(\frac{N}{5}) + T(\frac{7N}{10}) + N$$

Now we will solve this recurrence using induction. Our assumption will be $T(n) = O(n)$.
This leads to $T(n) \leq cn$
Induction case:

$$T(N) \leq c(\lceil \frac{N}{5} \rceil) + c(\frac{7N}{10}) + an$$

$$T(N) \leq c(\frac{N}{5}) + c(\frac{7N}{10}) + an$$

$$T(N) = 7c + c(\frac{9N}{10}) + an$$

$$T(N) = cN + \left(-\frac{cn}{10} + 7c + an\right)$$

The term in the brackets is less than 0 for $n > 140$. Therefore for $n > 140$, we get:

$$T(N) \leq cN$$

Base Case:

$$T(140) \leq T(\lceil \frac{140}{5} \rceil) + T(\frac{7 \times 140}{10}) + O(n)$$

$$T(140) \leq O(1) + O(1) + a \times 140$$

$$T(140) \leq b + b + 140a \leq 140c$$

where $c \geq a + \frac{b}{70}$ satisfies the condition. Choosing $c \geq 20a + b$ will satisfy base and inductive cases.

4.1.4 Evaluation

We use Valgrind software, to visualize the change in cache miss rates for different dimensions of array. We have plotted the cache miss rates as a function of array size in Fig. 4.3. It can be observed that with increase in array size, the cache miss rate for all cache types decreases.

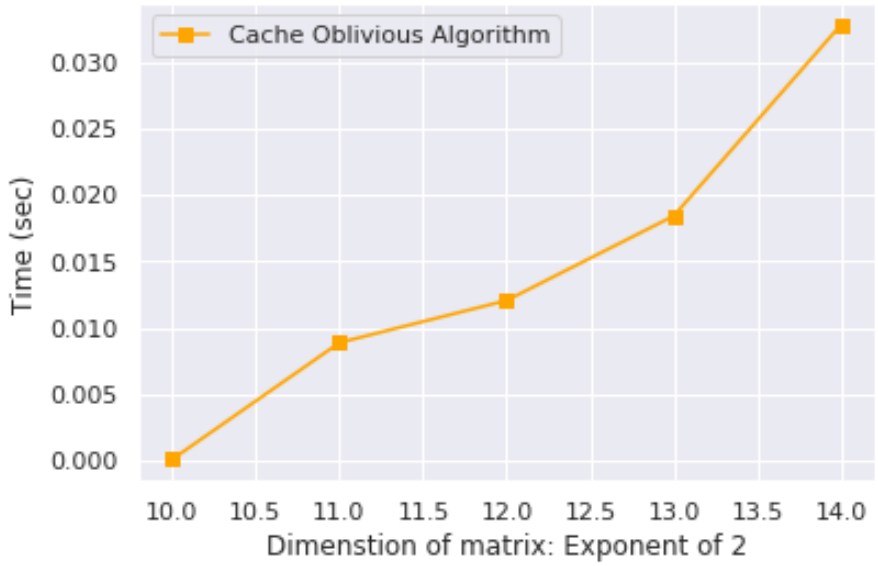

Figure 4.2: Computational time as a function of array dimension

Further, we also calculated the computational time as a function of array dimension, which has been plotted in Fig. 4.2.

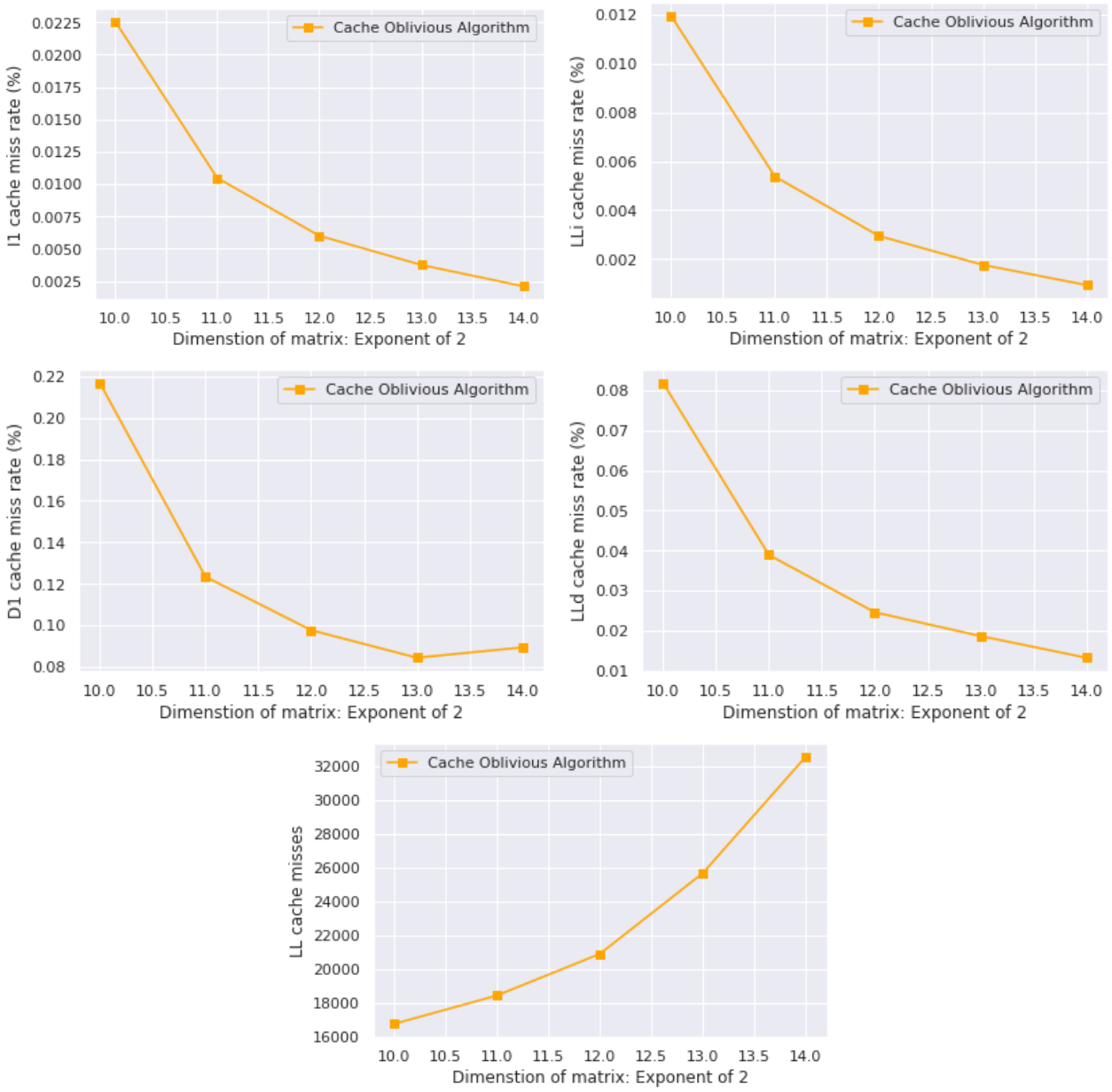

Figure 4.3: Plots showing miss-rates of cache oblivious median finding algorithm.

4.2 Matrix Transposition

4.2.1 Algorithm Design

The basic idea here is to use a recursive approach that repeatedly divides the data set until it becomes cache resident and hence cache friendly.

In the given pseudocode, A is matrix of order $n \times m$ which is to transposed into matrix B of order $m \times n$. Matrices are stored in a row major order.

Algorithm 4: Matrix Transposition

1: **procedure** TRANSPOSE(A, B) **if** $n = m = 1$ **then**
    $B_{11} \leftarrow A_{11}$
    else if mn **then**
    $k \leftarrow \lceil \frac{m}{2} \rceil$;
    Let $A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$, where A_1 has k rows and A_2 has n-k rows. ;
    Let $B = \begin{pmatrix} B_1 & B_2 \end{pmatrix}$, where B_1 has k columns and B_2 has n-k columns ;
    $Transpose(A_1, B_1)$;
    $Transpose(A_2, B_2)$;
    else
    $k \leftarrow \lceil \frac{n}{2} \rceil$;
    Let $A = \begin{pmatrix} A_1 & A_2 \end{pmatrix}$, where A_1 has k columns and A_2 has n-k columns. ;
    Let $B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$, where B_1 has k rows and B_2 has n-k rows ;
    $Transpose(A_1, B_1)$;
    $Transpose(A_2, B_2)$;
    end
2: **end procedure**

0	2	4	6	14	16	22	24
1	0	8	10	18	20	26	28
3	7	0	12	30	32	38	40
5	9	11	0	34	36	42	44
13	17	29	33	0	46	48	50
15	19	31	35	45	0	52	54
21	25	37	41	47	51	0	56
23	27	39	43	49	53	55	0

Figure 4.4: Access pattern for Matrix Transposition algorithm [8]

4.2.2 Memory Transfer Analysis

Theorem 4 *Memory transfer complexity of matrix multiplication of matrices of order $n \times m$ where n is a power of 3 is $O(1 + \frac{nm}{B})$.*

Proof: let α be such that for sub-matrices $n' \times m'$ and $m' \times n'$ completely fits in cache.
Now, if $maxn, m \leq \alpha B$, then total number of blocks to fit both matrices is $MT(O((1)) + \frac{nm}{B}$ and because of α, $MT(O(1))$ is small, hence $MT(n, m) = O(1 + \frac{mn}{B})$.
Else, we will break our matrix from its largest dimensions, till we have sub-matrices which fit into cache.
If, $m \geq n$ then, $MT(m, n) \leq 2MT(m/2, n) + O(1)$.
Else if If, $m \leq n$ then, $MT(m, n) \leq 2MT(m, n/2) + O(1)$.
Else if $max(m, n) \leq \alpha B$, then, $MT(m, n) = O(1 + \frac{mn}{B})$
This, recurrsion solves to $MT(m, n) = O(1 + \frac{mn}{B})$

4.2.3 Time Complexity Analysis

For a matrix of size $n \times m$, we have a problem $T(n \times m)$. Now, from the 4, we have,
If $n \geq m$, then $T(n \times m) = 2T(n/2 \times m) + O(1)$.
Else, $T(n \times m) = 2T(n \times m/2) + O(1)$.
In either case, we have $T(n \times m) = 2T(\frac{n \times m}{2}) + O(1)$ From master theorem, this solves to $T(n \times m) = O(n \times m)$.

4.2.4 Evaluation

Using Valgrind software, we plot the cache miss ratio versus the dimension of matrix for different cache levels and types. As seen from the Fig. 4.11, cache oblivious algorithm outperforms the cache-ignorant algorithm in all the 5 cases.

Further, we also calculated the computational time as a function of array dimension, which has been plotted in Fig. 4.6.

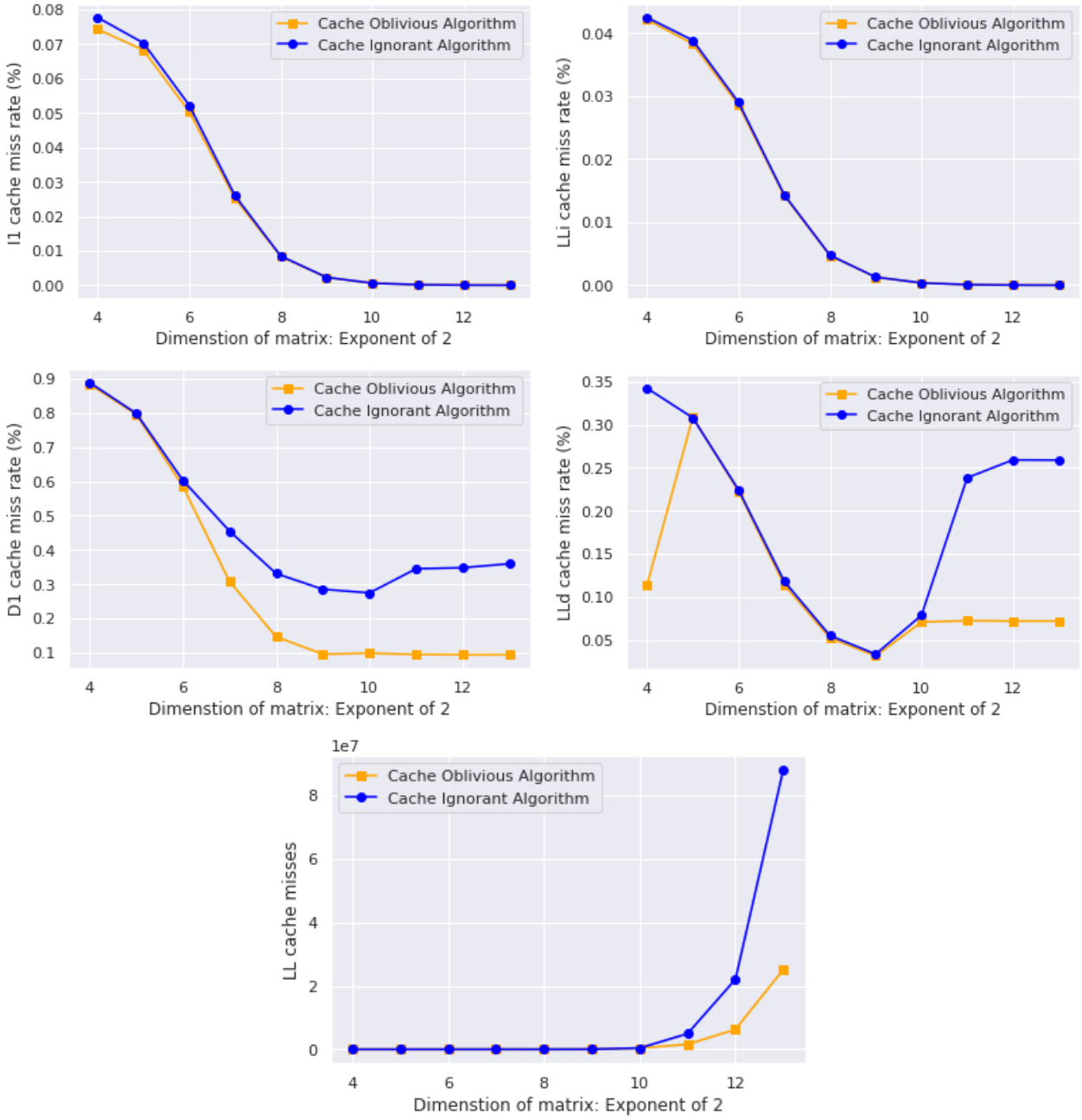

Figure 4.5: Plots comparing cache efficiency of cache oblivious matrix transposition with normal matrix transposition algorithm.

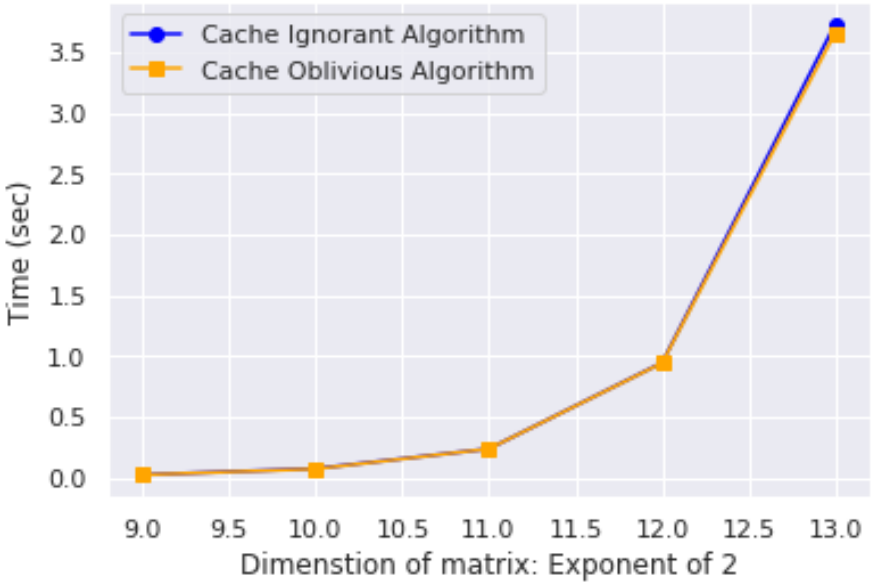

Figure 4.6: Computational time as a function of array dimension

4.3 Matrix Multiplication

4.3.1 Algorithm Design

Generally matrices are indexed in row-major or column-major order. But these methods of indexing are not good enough for cache efficient reasons due to large number of cache jumps. So we will use peano based indexing to store the matrix. Hereby is shown example of how peano indexing will be done of 3×3 and subsequently larger matrices.

(a) Peano indexing for 3×3 matrix

(b) Peano indexing for 9×9 matrix

Figure 4.7: Peano indexing. Indexing is done starting from tail to arrow [1]

You can see that Peano indexing for 9×9 matrix is done by using a 3×3 matrix recursively. Given below is the standard matrix multiplication technique.

Algorithm 5: Standard Matrix Multiplication Algorithm

```
for i ← 1 to n by 1 do
    for j ← 1 to n by 1 do
        C[i, j] ← 0 ;
        for k ← 1 to n by 1 do
            C[i, j] ← C[i, j] + A[i, k] × B[k, j] ;
        end
    end
end
```

Now this algorithm can also be written in a more lenient way as follows. Here all valid triples are the triples which forms pairs in matrix multiplication.

Algorithm 6: Standard Matrix Multiplication Algorithm(revised)

```
while all valid triples in (i, j, k) do
    C[i, j] ← C[i, j] + A[i, k] × B[k, j]
end
```

Now this new 6 is very helpful as in this algorithm, multiplication can be performed in any order due to commutativity. From the given figure below we can see and note all the valid

triples for matrix multiplication.

$$\underbrace{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}}_{=:B} = \underbrace{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}}_{=:C}$$

Figure 4.8: Matrix Multiplication in Peano indexed matrices [1]

Now to avoid cache jumps, we want to choose our triples for matrix multiplication in such a way that for all triples (i, j, k), difference between indexes of any components, in adjacent nodes is not greater than 1. This scheme helps in finding optimum serialization and reducing number of memory transfers as we can directly reuse an element or move to its neighbour after each operations.

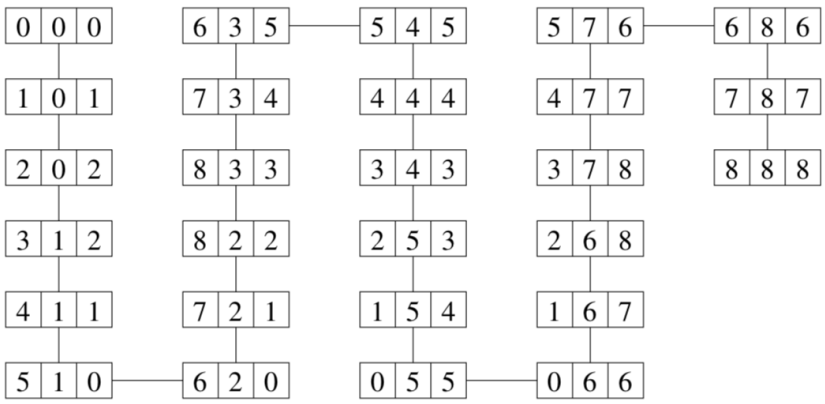

Figure 4.9: Matrix Multiplication Scheme for 3×3 matrix, triples are of form(i, j, k) [1]

For matrices of dimensions larger than 3×3, we will apply this given scheme using divide and conquer.

Now given is the function which uses this multiplication scheme to find do matrix multiplication.[1]

```
int a = 0, b = 0, c = 0;
vector<int> A, B, C;

/* global variables
 * A, B, C: the matrices, C will hold the result of AB
 * a, b, c: indices of the matrix element of A, B, and C
 */

void peanomult(int phsA, int phsB, int phsC, int dim){
    if(dim == 1){
        C[c] += A[a]*B[b];
    }
    else{
        peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; b += phsB;

        peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; c -= phsC;
        peanomult( phsA, -phsB, -phsC, dim/3); a += phsA; c -= phsC;
        peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; b += phsB;

        peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA,  phsB,  phsC, dim/3); b += phsB; c += phsC;

        peanomult(-phsA,  phsB,  phsC, dim/3); a -= phsA; c += phsC;
        peanomult(-phsA, -phsB,  phsC, dim/3); a -= phsA; c += phsC;
        peanomult(-phsA,  phsB,  phsC, dim/3); a -= phsA; b += phsB;

        peanomult(-phsA,  phsB, -phsC, dim/3); a -= phsA; c -= phsC;
        peanomult(-phsA, -phsB, -phsC, dim/3); a -= phsA; c -= phsC;
        peanomult(-phsA,  phsB, -phsC, dim/3); a -= phsA; b += phsB;

        peanomult(-phsA,  phsB,  phsC, dim/3); a -= phsA; c += phsC;
        peanomult(-phsA, -phsB,  phsC, dim/3); a -= phsA; c += phsC;
        peanomult(-phsA,  phsB,  phsC, dim/3); b += phsB; c += phsC;

        peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; b += phsB;

        peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; c -= phsC;
        peanomult( phsA, -phsB, -phsC, dim/3); a += phsA; c -= phsC;
        peanomult( phsA,  phsB, -phsC, dim/3); a += phsA; b += phsB;

        peanomult( phsA,  phsB,  phsC, dim/3); a += phsA; c += phsC;
        peanomult( phsA, -phsB,  phsC, dim/3); a += phsA; c += phsC;
```

```
48            peanomult( phsA,  phsB,  phsC,  dim/3);
49        }
50    }
```

4.3.2 Memory Transfer Complexity

Theorem 5 *Memory transfer complexity of matrix multiplication of matrices of order $n \times n$ where n is a power of 3 is $O(\frac{(N^3)}{B\sqrt{M}})$.*

Proof: In the algorithm given above we can see that for each function of dimension N, there are 27 recursive calls of $dimension = \frac{N}{3}$.
Therefore, $MT(N) = 27MT(\frac{N}{3}) = (3^3)MT(\frac{N}{3})$. Now, let n be the largest power of 3, such that three $n \times n$ matrices fit into the cache.
Hence, $3n^2 \leq M$, but $3(3n)^2 > M$, or

$$\frac{1}{3}\sqrt{\frac{M}{3}} < n < \frac{1}{3}\sqrt{\frac{M}{3}}$$

Now,

$$MT(N) = (3^{3k})MT(\frac{N}{k})$$

Now this recursion will occur till $\frac{N}{k}$ is stored in cache memory, or

$$\frac{N}{3^k} = n, MT(N) = (\frac{N}{n})^3 MT(n)$$

For a $n \times n$ sub-matrix, matrix will fir in $1 + \lceil \frac{n^2}{B} \rceil$ lines and hence this number of cache-tranfers will be required. As there are two matrices of $n \times n$ that are to be processed, therefore, $MT(n) = 2\lceil \frac{n^2}{B} \rceil$ and hence,

$$MT(N) = (\frac{N}{n})^3 \times 2(\lceil \frac{n^2}{B} \rceil) \leq (\frac{N}{\frac{1}{3}\sqrt{\frac{M}{3}}})^2.2(\frac{n^2}{L} + 1)$$

Therefore,

$$MT(N) = O(\frac{N^3}{L\sqrt{M}})$$

4.3.3 Time Complexity Analysis

From the code for function of matrix multiplication given above, we can see that for dimension N, there are 27 recursive calls of $dimension = \frac{N}{3}$.
Therefore, $T(N) = 27T(\frac{N}{3}) + O(1) = (3^3)T(\frac{N}{3}) + O(1)$.
Using Master's theorem here, we get $T(N) = O(N^3)$

4.3.4 Evaluation

Using Valgrind software, we plot the cache miss ratio versus the dimension of matrix for different cache levels and types. As seen from the Fig. 4.11, cache oblivious algorithm performs better or equally good as the cache-ignorant in all the 5 cases.

Further, we also calculated the computational time as a function of array dimension, which has been plotted in Fig. 4.10.

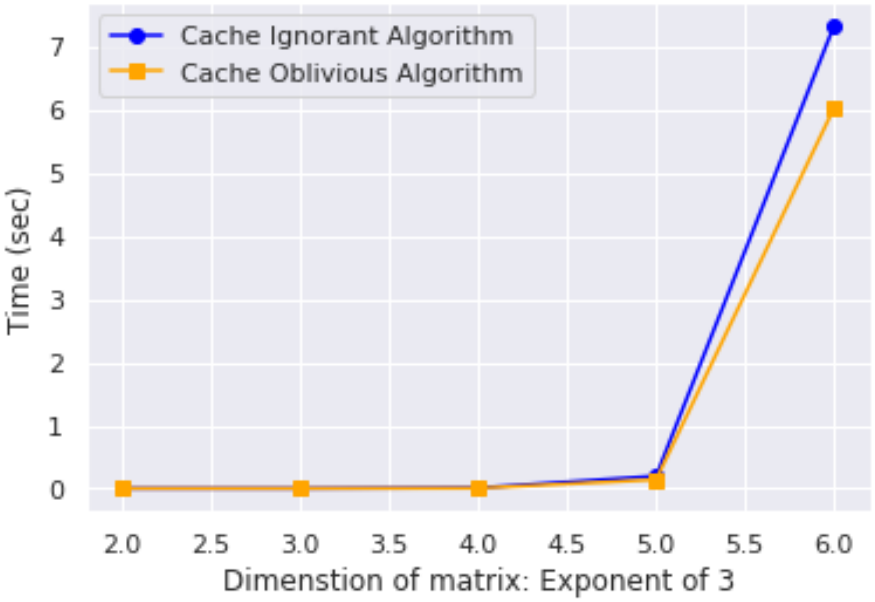

Figure 4.10: Computational time as a function of matrix dimension

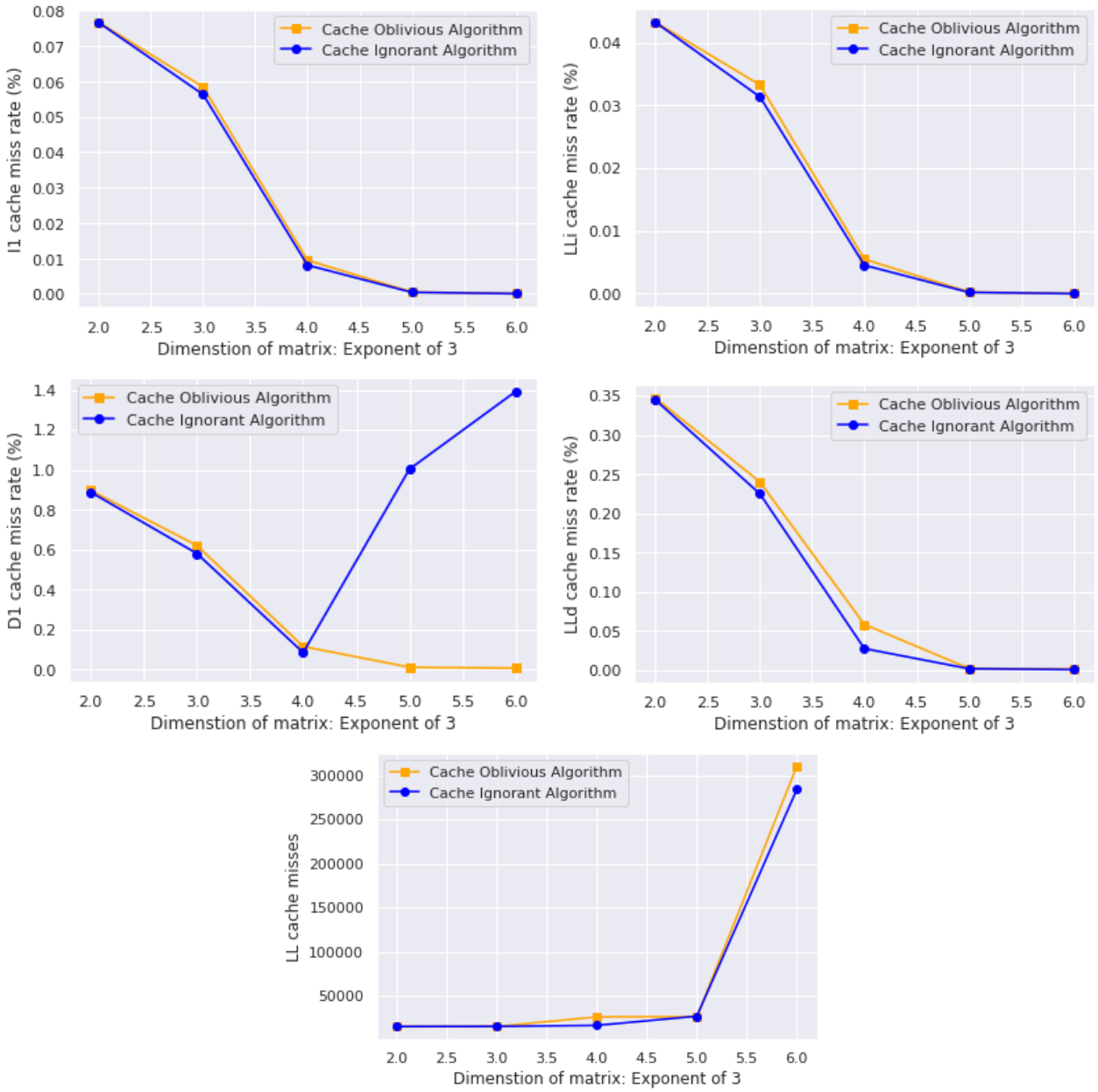

Figure 4.11: Plots comparing cache efficiency of cache oblivious matrix multiplication with normal matrix multiplication. Cache oblivious version performs better on D1 cache miss rates, and equally well in other miss-rates at higher dimensions.

4.4 Van Emde Boas Static Search Tree

4.4.1 Algorithm Design

Traditional binary search algorithm requires $O(log\frac{N}{B})$ memory transfers. This can be significantly improved to $O(log_B N)$ memory transfers using Van Emde Boas static search tree. For we first convert the sorted array on which we have to search in Van Emde Boas layout using algorithm 7. For better understanding, the recursive layout has been shown in Fig. 4.12.

Algorithm 7: Van Emde Boas layout algorithm [5]

1: Layout the elements of array in Binary search tree format.
2: Conceptually split the tree at the middle level of edges, resulting in one top recursive subtree and roughly $\sqrt{N}$ bottom recursive subtrees, each of size approximately $\sqrt{N}$.
3: Recursively lay out the top recursive subtree, followed by each of the bottom recursive subtrees.

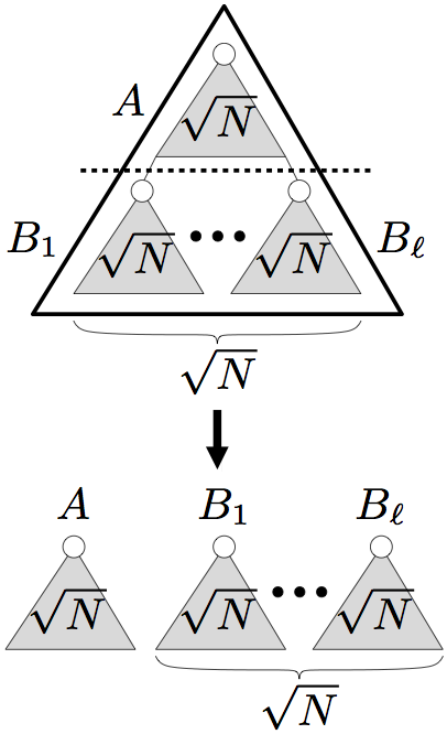

Figure 4.12: Illustration of Van Emde boas layout [5]

4.4.2 Memory transfer complexity

Theorem 6 *Memory transfer complexity of searching an element in Vam Emde Boas static search tree is* $O(log_B N)$.

Proof: Searching algorithm in Van Emde boas layout is same as searching in Binary search tree layout - starting from the root and moving downwards in the tree till the element is found (or not found). An example traversal is shown in Fig. 4.13. The height of base-case recursive sub-tree is size $O(logB)$. The height of the complete tree is $O(logN)$. Therefore, number of memory transfers in going from root to leaf is $O\left(\frac{logN}{logB}\right) = O(log_B N)$.

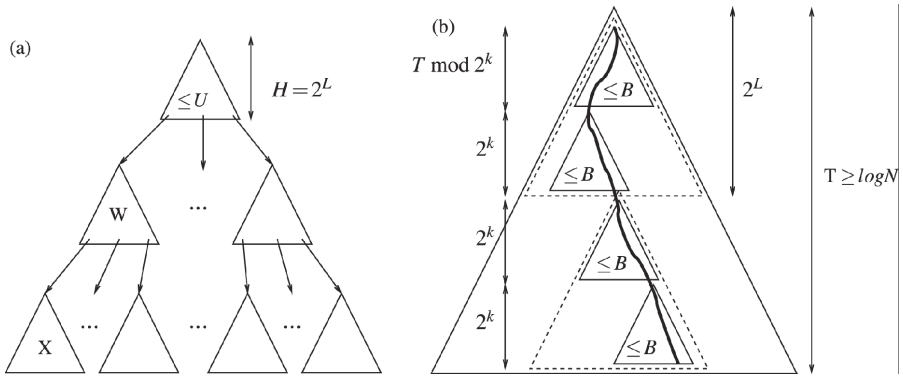

Figure 4.13: Illustration of traversal in Van Emde Boas Static search tree. [5]

4.4.3 Time Complexity Analysis

Time complexity analysis for this case is trivial. It is same as that of searching an element in Binary Search Tree, i.e., $O(logN)$. This is because, here also we start from the root, and move towards the leaves in search of the element. The algorithm of searching an element in Van Emde Boas Search tree is same as that of Binary Search Tree, and hence the time complexity is also same. The difference lies in memory layout, which gives superior memory transfer complexity.

4.4.4 Evaluation

Using Valgrind software, we plot the cache miss ratio versus the size of array for different cache levels and types. As seen from the Fig. 4.14, Van Emde Boas static search tree layout performs significantly better than binary search in all the 5 cases.

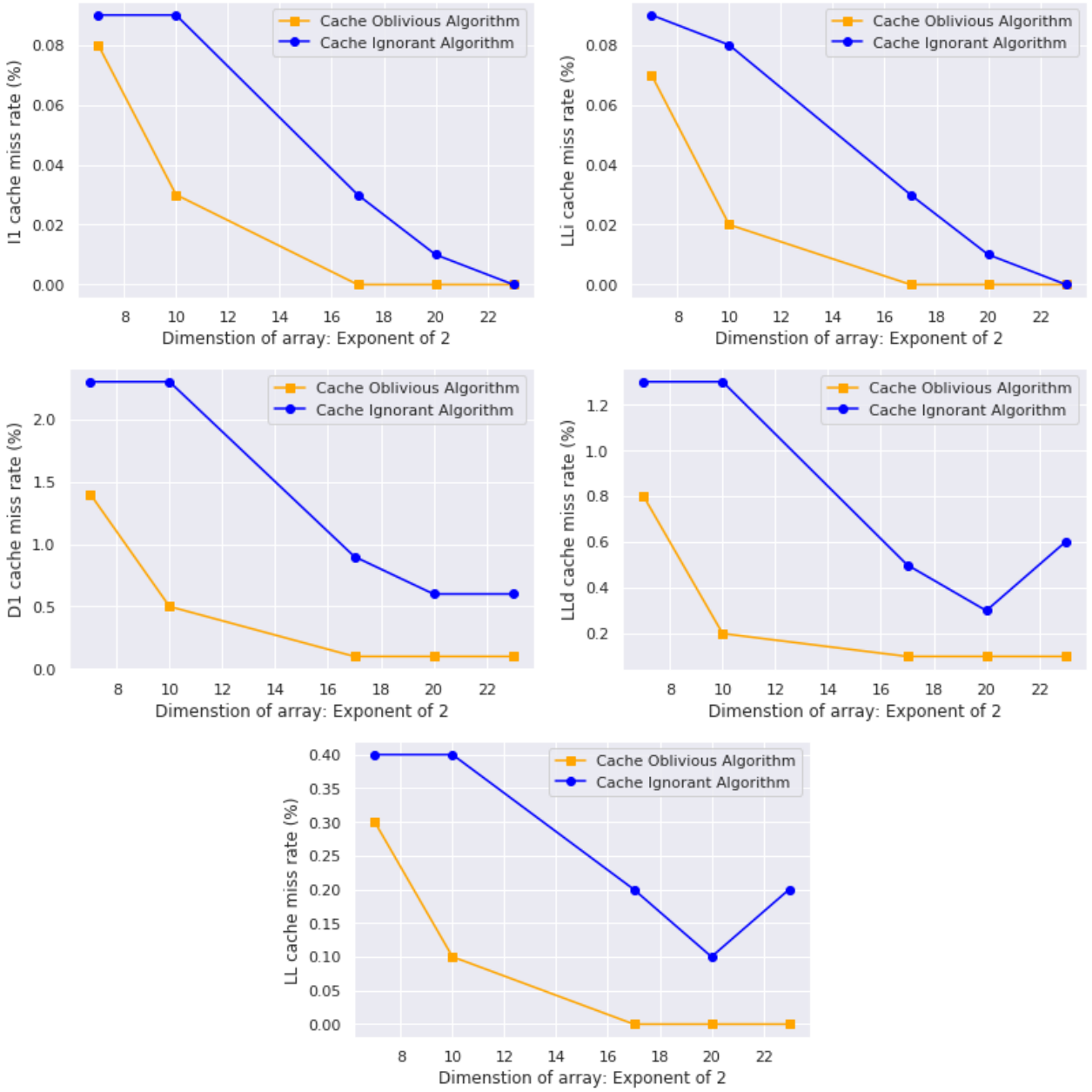

Figure 4.14: Plots comparing cache efficiency of Van Emde Boas static search tree with normal binary search algorithm. Van Emde Boas static search tree performs significantly better in all 5 cases.

4.5 Funnel Sort

4.5.1 Algorithm Design

Funnel sort is basically a $N^{1/3}$-way merge sort. It attains the optimal memory transfer complexity of $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$, provided the tall cache assumption $M = \Omega(B^{1+\gamma})$ holds. It also attains time complexity of $O(nlogn)$. For simplicity, we take $M = \Omega(B^2)$. The same results can be obtained when $M = \Omega(B^{1+\gamma})$. The basic steps of funnel-sort is shown in algorithm 8:

Algorithm 8: Funnel sort algorithm [3]

1: Divide the array into $K = N^{1/3}$ continuous segments. Hence, each segment is of the size $N/K = N^{2/3}$.
2: Use recursion to sort these $N^{2/3}$ segments.
3: Use the K-funnel data-structure to merge these $N^{2/3}$ segments.

The merging step is done by a data-structure called a k-merger, which inputs k sorted sequences and merges them, as illustrated in Fig. 4.15.

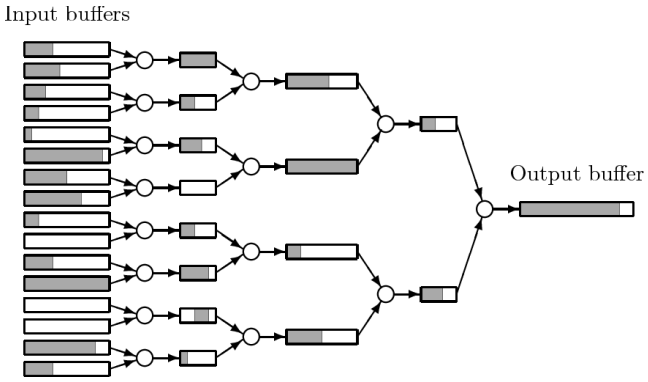

Figure 4.15: Illustration of merging by k-merger data-structure [3]

K-funnel can merge K sorted list of total size K^3 in $O\left(\frac{K^3}{B}\log_{M/B}\frac{K^3}{B} + K\right)$ [5]. The k-merger algorithm is shown in algorithm 9.

Algorithm 9: Algorithm for k-merger [5]

while *output buffer is not complete* **do**
  if *left input buffer is empty* **then**
    Recurse on left child to fill it's input buffer
  else
    Recurse on right child to fill it's input buffer
  Merge the input buffers of left and right child to fill the output buffer.

4.5.2 Memory transfer complexity

Theorem 7 *Assuming the tall cache assumption $M = \Omega(B^2)$ holds, the memory transfer complexity of funnel sort is $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ [4].*

Proof: Seeing steps 2 and 3 of algorithm 8, the following memory transfer recurrence can be obtained:

$$MT(N) = N^{1/3}MT(N^{2/3}) + O\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + N^{1/3}\right)$$

Since the tall cache assumption $M = \Omega(B^2)$ holds, we can use the base case is $MT(O(B^2)) = O(B)$. This is because this whole data can fit inside the cache and hence no more memory transfers are required. Since, $N = \Omega(B^2)$, therefore B=$O(\sqrt{N})$. Therefore $N^{1/3}$ term is polynomially smaller than $\frac{N}{B}\log_{M/B}\frac{N}{B}$, and therefore can be ignored.

On opening the recursion tree and summing all the levels, we get $MT(N) = O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$.

4.5.3 Evaluation

In order to show efficacy of funnel sort in terms of number of memory transfers, we compare it with quick sort algorithm. We use valgrind software to compute cache miss ratio. We have shown 5 plots of different cache types and levels in Fig. 4.16. In every case, funnel-sort outperforms quicksort.

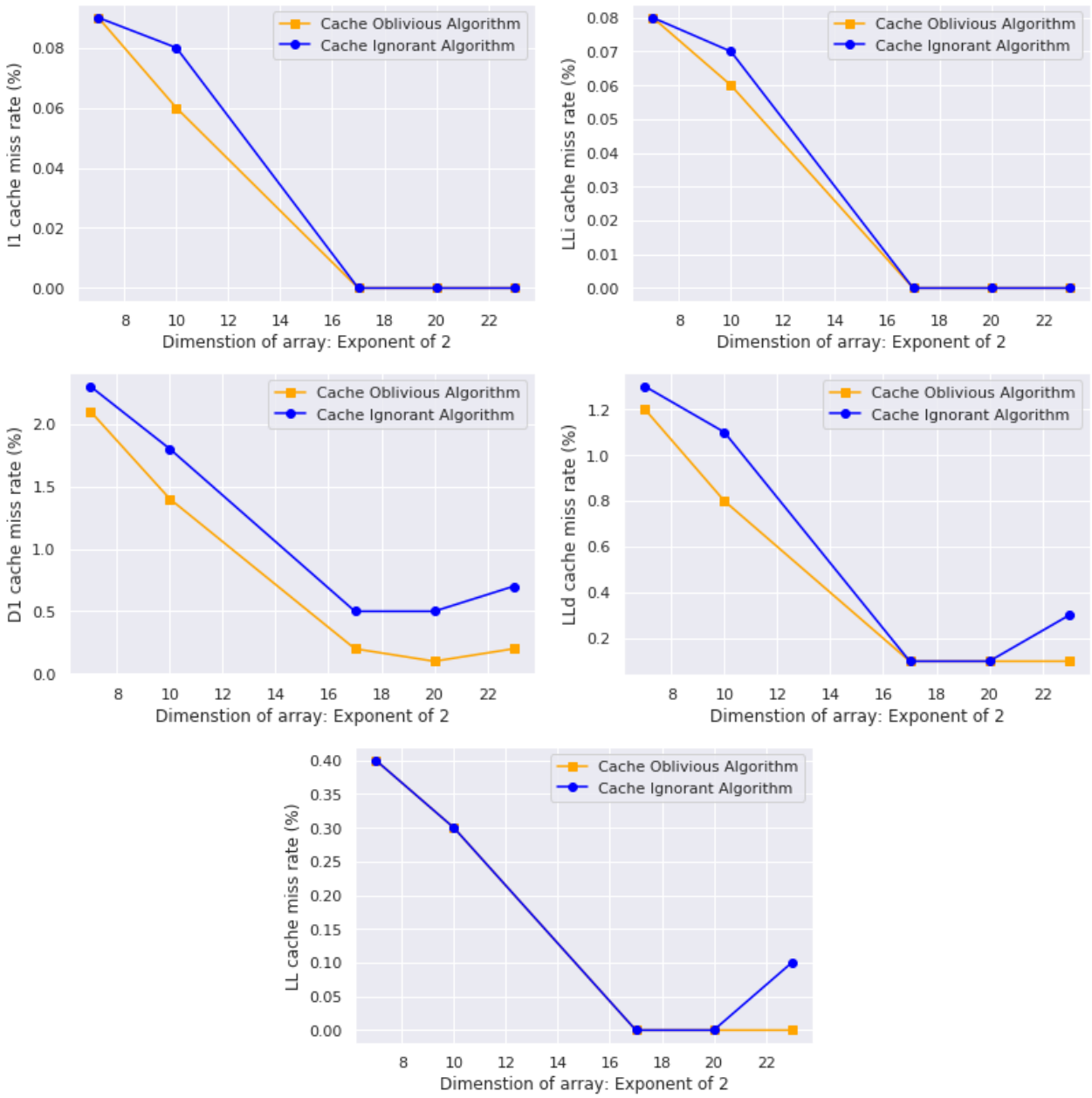

Figure 4.16: Plots comparing cache efficiency of funnel-sort with quicksort algorithm. Funnel-sort performs better in all the cases, especially in D1 cache miss ratio.

5. Conclusion and Future Directions

The cache-oblivious model remains an promising area of research that has prompted many researchers to work in this domain. There has been recently a lot of focus of the academia into design of cache oblivious algorithms and data structures. It promises to significantly improve the memory transfer complexities of algorithms which has proven to be a severe bottle neck in past.

Our analysis in this project sought to reproduce these results for various algorithms, for which we think we were successful. Using Valgrind software, we have plotted cache miss hit ratio for various algorithms, clearly showing superior performance of cache oblivious algorithms with respect to cache ignorant algorithms.

As a part of future work we aim to implement and do critical analysis of cache-oblivous dynamic data structures such as priority queues, ordered-file maintenance etc.

Bibliography

[1] Michael Bader and Christoph Zenger. Cache oblivious matrix multiplication using an element ordering based on a peano curve. *Linear Algebra and Its Applications*, 417(2-3):301–313, 2006.

[2] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 189–199, 2010.

[3] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. Engineering a cache-oblivious sorting algorithm. *Journal of Experimental Algorithmics (JEA)*, 12:1–23, 2008.

[4] Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

[5] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms (TALG)*, 8(1):1–22, 2012.

[6] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[7] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

[8] Dimitrios Tsifakis, Alistair P Rendell, and Peter E Strazdins. Cache oblivious matrix transposition: Simulation and experiment. In *International Conference on Computational Science*, pages 17–25. Springer, 2004.