Interpretation & Conclusion
A Cache memory is high speed and relatively small memory which play very important role in computer system. It is utilized to eliminate the gap between CPU and main memory speed. It has a smaller access time than that of main memory. At present there are three types of cache

1. Fully Associative
2. Direct Mapped
3. Set Associative

In Fully Associative Cache, there is no restriction on mapping form memory to cache. The tag used for the search it is expensive. It is feasible for small size cache only. In Direct Mapped Cache, A memory block is mapping by only one cache line. There is no need to expensive search. In Set Associative, A predefined set are used for mapping from memory to cache. From this list of types of Cache, Direct Mapped cache is the fastest. With advance development in hardware of computer, it is possible to have high speed CPU but the performance of CPU get restricted by the performance of Cache memory access. There are various approaches available for effective utilization of cache memory. List Recently Used (LRU), Most Recently Used (MRU), Pseudo LRU (PLRU), Segmented LRU, Set Associative Direct Map, Least Frequency Used (LFU) and Multi Queue (MQ).
In first chapter we introduce cache memory along with hardware and software approach. A designing algorithm with cache consideration significantly improves their performance. This is most noticeable in the more CPU intensive algorithms. We discuss historical review of development of hardware processing with utilization of memory and its improvements. For most of the 80s and 90s software compatibility motivated building machines to make a CPU faster. Given this environment, it is no surprise that industry responded by making CPUs faster building machines.

We give cache aware and cache oblivious environment of cache behaviors, it gives idea of cache techniques and its execution. We elaborate some commonly used design tools that are used to design cache algorithms. In summary, the main memory should not be the end of optimization for algorithms. Every algorithm which is designed and implemented it successfully run in its specific area only it will not used in different applications of computer systems. Designing algorithms that exploit the cache has significant performance dividends and this is becoming increasingly important for the newer generation of microprocessors whose performance critically depend upon effective usage of cache memory. It would be interesting to study how algorithms can exploit specific cache configurations and conversely what cache configurations are better suited for applications.

In second chapter we introduces new software SSCMU (Software System for Cache Memory Utilization) describes the main features that have included the implementation of a software cache implementation for various cache oblivious algorithms, in order to maximize the chances for computation and communication. We have developed
visualization interface to show the detailed performance of SSCMU in comparison with different cache oblivious algorithms. It automatically optimizes the performance of all levels of memory hierarchy. This automaticity is desirable for levels for L1 and L2 caches of main memory. As a result, the accurate state information of these caches is difficult to obtain due to the system runtime dynamics and the hardware complexity. We consider cache oblivious algorithms that can automatically improve the in memory performance of processing of matrix multiplication, matrix transposition, dynamic programming, Sorting, B-tree.

Hardware implements cache as a block of memory for temporary storage of data likely to be used again. In third chapter Cache Oblivious Algorithm is typically analyzed using an idealized model of cache, but it is much easier to analyze than a real cache memory. The memory of modern computer is layered in a hierarchy, top to bottom primary cache, secondary cache, main memory, virtual memory and distributed memory with more levels to come in the future. Our goal is to automatically achieve that of fine-tuned algorithms on a multi-level memory hierarchy. This automatically is because cache oblivious algorithms no knowledge about any capacity and block size of each level of the hierarchy. Efficient techniques are proposed to manage cache memory. The new techniques uses block recursive structure of two types only. The algorithm is tested on famous problem of matrix multiplication. It avoids jumps and cache misses are reduced to the order of $N^3/L\sqrt{M}$.

We present here the problem of optimization of cache memory done by implementation of optimal oblivious matrix multiplication. Our algorithm uses only two
types of recursive blocks. All the elements are accessed sequentially and there are no jumps at all. The number of cache miss are of the order of $O(N3/L \sqrt{M})$.

In fourth chapter we have presented four standard algorithms to yield Matrix Transposition algorithm from the same cache-oblivious algorithms on different array. The optimal cache oblivious transposition makes $O(n^2/B)$ cache misses. In new algorithm Sequential Processing for Matrix Transposition Algorithms it stores the element in Column-major order so it process element vice-verse of other elements so we eliminate the cache misses.

It gives $3\times3$, $4\times4$ and $5\times5$ square of matrices which totally eliminate the cache misses in matrix transposition algorithms.

As we pointed out, the algorithm can be generalized to the transposition of square of matrices of arbitrary size. If the number of rows and columns of the matrices are same $2$-by-$2$, $3$-by-$3$ for example we repeated up to matrix blocks of size $[x,y]$ where $x,y = \{2,3,4,5,\ldots,n\}$ of course the number of matrices has to be changed accordingly.

It should be possible to generalize the transposition scheme to certain different types of matrix schemes.

In fifth chapter we presented Sequential processing for sorting algorithm that has better locality features. Using ideal cache model cache misses is of order of $O\left(\frac{N}{B} \log_{MB} \frac{N}{B}\right)$ block transfers and permuting an array requires $O\left(\min\left\{\frac{N}{B} \log_{MB} \frac{N}{B}, \frac{N}{B}\right\}\right)$ block transfers. This is asymptotically optimal for any algorithm that is based on sorting. It
totally avoids the need for swapping for address arithmetic. While this fact is not fully exploited on standard computers, it may be considerable advantage for hardware implementations of sorting techniques.

Modern Computers contain a hierarchy of memory levels, with each levels acting as cache for the next. As a consequence the memory access pattern of an algorithm has a major influence on its running time in practices. This method is an algorithmic engineering study of cache oblivious sorting. We present cache oblivious algorithm for sorting. The algorithm uses array structure that is based on sorting methods. In resulting code it avoid swapping in sorting.

To summarize, we have studied cache oblivious algorithms for matrix multiplication, matrix transposition, dynamic programming, sorting and binary searching trees. The Algorithm design is used heavily in both parallel and external memory algorithms. Cache oblivious algorithms make heavy use of this paradigm and lot of seemingly simple algorithm that were based on this paradigm are already cache oblivious for instance, stressen’s matrix multiplication, quick sort, merge sort, closed pair, Convex hulls median selection are all algorithms are that are cache oblivious, through not all of them are optimal in this model. This means that they might be cache oblivious but can be modified to make fewer cache misses than they do in the current form.

The design of cache-oblivious requires a deep understanding of program structures and operation and familiarity with a machine’s memory architecture. Based on past trends and future technology, the processor-memory gap will continue increase and
software will continue to grow larger and more complex. Although the algorithmic and data structure design phase of software development is the first, and perhaps best, place to address this growing gap, the complexity of software design, and an increasing tendency to build large software systems by gluing together smaller components does not favor a focused, integrated approach. It is very essential to develop the process of achieving the highest performance on current and future machines.