INFLUENCE OF CACHES ON THE PERFORMANCE OF INSERTION SORT

By

SUNIL MEHTA

Bachelor of Engineering

University of Bombay

Bombay, India

1998

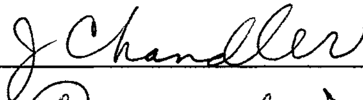
Submitted to the faculty of the
Graduate College of the
Oklahoma State University
In Partial Fulfillment of
The Requirements of
The Degree of
MASTER OF SCIENCE
May, 2003

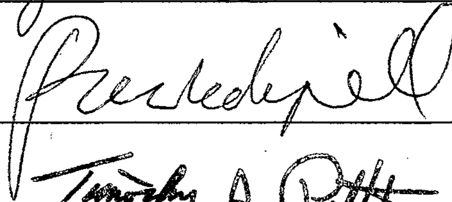
INFUENCE OF CACHES ON THE PERFORMANCE OF

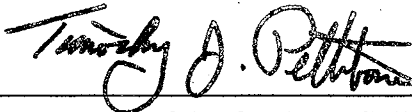
INSERTION SORT

Thesis Approved:

Thesis Advisor







Dean of the Graduate College

PREFACE

Sorting is used in many important applications; consequently, there has been an abundance of performance analyses of sorting algorithms. However, most previous research does not take into account memory hierarchies present in today's computers. Since most computers today contain cache storage, it is important to analyze sorting methods based on their cache performance. Due to the increase in the cache miss penalty, the relative performance results we obtain on today's machines greatly differ from the machines of ten years ago. Recent research in this area has proved that caches affect the performance of sorting algorithms, in comparison to their non-cached architectures. Current research in this area concentrates mainly on mergesort, quicksort, heapsort and radixsort. In this thesis the effect of caches on the performance of insertion sort is investigated and closed form solutions for its miss rate are developed analytically. Simulations are used to verify these analytical solutions. Finally, its traditional theoretical complexity is compared to its cache based performance.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. G. E. Hedrick, my major adviser, for his constant inspiration and encouragement throughout my graduate study. None of this would have been possible without his consistent advice and generous aid. I also wish to express my sincere gratitude to Dr. J. P. Chandler and Dr. N. Park for their guidance and support during my graduate study.

I am grateful for the help I received from many individuals. In particular, I wish to extend my gratitude to Dr. R. Sharda and Dr. M. Weiser for their kind help.

Last, but not least, my appreciation goes to my parents, Rita Mehta and Shillander Mehta, and my brother, Anil Mehta, for their constant love, encouragement and support.

TABLE OF CONTENTS

Chapter		Page
I	INTRODUCTION	1
1.1	Sorting Algorithms	1
1.2	Theme of the Thesis	2
1.3	Thesis Organization	3
II	LITERATURE REVIEW AND BASIC CONCEPTS	4
2.1	Brief Review	4
2.2	Caches and Data Locality	6
2.3	Caches and Algorithm analysis	7
III	CACHE BASED ANALYSIS OF INSERTION SORT	10
3.1	Insertion Sort	10
3.2	Cache Based Analysis of Insertion Sort	11
	3.2.1 Insertion Sort Source Code	11
	3.2.2 Best Case Analysis	13
	3.2.3 Worst Case Analysis	14
	3.2.4 Average Case Analysis	18
IV	SIMULATION AND COMPARATIVE GRAPHS	21
4.1	Simulation	21
4.2	Graphs	22
4.3	Observations	28

Chapter		Page
V	CONCLUSION AND FUTURE WORK..	29
BIBLIOGRAPHY...		30
APPENDICES...		32
	APPENDIX A- GLOSSARY..	33
	APPENDIX B- SIMULATION PROGRAM SOURCE CODE...........	34

LIST OF FIGURES

Figure		Page
1	RAM Model	9
2	Memory hierarchy of Modern computers	9
3	Insertion Sort Source code	11
2	Best Case Analysis	13
3	Inverse Sorted Array with Data Elements	15
4	Worst Case Analysis (Pass Less Than Cache Size)	16
5	Worst Case Analysis (Pass Greater Than Cache Size)	16
6	Average Case Analysis (Pass Less Than Cache Size)	19
7	Average Case Analysis (Pass Greater Than Cache Size)	19

Chapter 1

Introduction

1.1 Sorting algorithms

Sorting is a fundamental task that is performed by most computers. It is used frequently in a large variety of important applications. All *spreadsheet programs* contain some sort of sorting code. *Database applications* used by schools, banks, and other institutions all contain sorting code. Because of the importance of sorting in these applications, dozens of sorting algorithms with varying complexity have been developed over the decades.

Varying in complexity, sorting algorithms fall into two basic categories –

a. Comparison based: A comparison-based algorithm orders an array by weighing the value of one element against the value of other elements. Algorithms such as quicksort, mergesort, heapsort, bubble sort, and insertion sort are comparison based.

b. Non-comparison based : Alternatively, a non-comparison based algorithm sorts an array without comparing pair-wise data elements. Radix sort is a non-comparison based algorithm that treats the sorting elements as numbers represented in a base-M number system, and then works with individual digits of M.

Comparison based sorting algorithms can be categorized based on their theoretical time complexity. Slow sorting methods such as bubble sort, insertion sort, and selection sort have a theoretical time complexity of $O(N^2)$ in average and worst case. Shellsort, which

is based on insertion sort, was one of the first algorithms to break the quadratic barrier. Even though these algorithms are very slow for sorting large arrays, each algorithm is logically simple, so they are not useless. If an application only needs to sort moderately large arrays, then it is satisfactory to use one of the simple slow sorting algorithms as opposed to a faster, but more complicated sorting algorithm. For these applications, the increase in coding time and probability of a coding mistake in using the faster sorting algorithm is not worth the speedup in execution time.

1.2 Theme of this thesis

Sorting is used in many important applications; consequently, there has been an abundance of performance analyses of sorting algorithms. However, most previous research does not take into account memory hierarchies present in today's computers. Since most computers today contain cache storage, it is important to analyze sorting methods based on their cache performance. Due to the increase in the cache miss penalty, the relative performance results we obtain on today's machines greatly differ from the machines of ten years ago. Recent research in this area has proved that caches affect the performance of sorting algorithms, in comparison to their non-cached architectures. Current research in this area by LaMarca [5,6,7] and Ladner [6,7] concentrates mainly on mergesort, quicksort, heapsort and radixsort. In this thesis the effect of caches on the performance of insertion sort is investigated and closed form solutions for its miss rate are developed analytically. Simulations are used to verify these analytical solutions. Finally, the traditional theoretical time complexity of insertion sort is compared to its cache based performance.

In this thesis we focus on insertion sort, and study the influence of cache performance on its time complexity. Then closed form solutions for its miss rate are proposed. These solutions are verified using simulations. We further compare its theoretical complexity with its cache-based performance giving performance curves.

1.3 Thesis Organization

This thesis is organized in the following way: chapter II provides the literature reviews of the basic concepts that appear in the thesis; what other people did in the area of cache analysis of algorithms and an overview of caches and algorithm analysis. Chapter III describes in detail the cache based analysis of Insertion sort for the best, worst and average cases, giving closed form solutions for the number of cache misses in each case. Chapter IV briefly describes the simulation program and gives comparative graphs. Chapter V summarizes the effects of caches on the performance of insertion sort and describes possible future work in this area.

Chapter 2

Literature Review and Basic Concepts

2.1 Brief Review

Most previous research is based on the algorithms' theoretical complexity using a non-cached architecture. The performance analysis of algorithms mostly was based on the theory behind the algorithm. Since most computers today contain a cache, it is important to analyze them based on their cache performance. Donald Knuth [3] has studied many sorting algorithms in great detail. However, even though Knuth gives a complete analysis of the different algorithms, they are all based on a non-cached computer architecture. All of his analyses are based on the theoretical complexity of the algorithms.

As the cached computer architecture becomes common today, it becomes desirable to analyze how a cached memory affects the performance of these sorting algorithms. Theoretical analyses are still useful because they are the fundamental analyses that are needed in analyzing any kind of algorithm. Even though there is an abundance of previous research on the performance of sorting algorithms, most of the research does not analyze how the sorting algorithms exploit caches. Since almost all of today's computers contain a cached memory architecture, this is an area that is lacking in research. In addition, as the increase in memory access time becomes larger than the increase in processor cycle time, then the cache performance of an algorithm has an increasingly larger impact on the overall performance.

Loop tiling, or blocking, has been used effectively to reduce cache miss rates. Preeti Ranjan Panda et al. [9] and Monica S. Lam et al. [4] have studied the performance of blocking algorithms in great detail. These fundamentals can be applied to various sorting algorithms to improve their overall performance by reducing their cache miss rate.

Lately there has been an increased awareness in analyzing performance of sorting algorithms taking into account caches and locality rather than analyses based on traditional theoretical complexity. Current research on analyzing and reducing the cache miss rate of algorithms is attributed to LaMarca [5,6,7]. He carried out a detailed study of the influence of caches on sorting algorithms [5,6]. His study mainly focused on analyzing and improving mergesort, quicksort and heapsort by reducing their cache miss rates. He has also presented closed form solutions for the miss rates of these algorithms. Influence of caches on heaps was studied in detail by LaMarca and Ladner[7]. Recent research by Ying Shi and Eushiun Tran [11] has further proved that cache does affect the performance of these sorting algorithms. Alpha sort, which is a new cache-sensitive memory-intensive parallel sort algorithm, was studied by Chris Nyberg et al [8].

2.2 Caches and Data Locality

In order to speed up memory accesses, small high speed memories called caches are placed between the processor and the main memory. Accessing the cache is typically much faster than accessing main memory. Unfortunately, since caches are smaller than main memory they can hold only a subset of its contents. Memory accesses first consult the cache to see if it contains the desired data. If the data is found in the cache, the main memory need not be consulted and the access is considered to be a cache hit. If the data is not in the cache it is considered a miss, and the data must be loaded from main memory. On a miss, the block containing the accessed data is loaded into the cache in the hope that it will be used again in the future. The hit ratio is a measure of cache performance and is the total number of hits divided by the total number of accesses.

The major design parameters of caches are:

1. **Capacity:** which is the total number of bytes that the cache can hold.
2. **Block size:** which is the number of bytes that are loaded from and written to memory at a time.
3. **Associativity:** which indicates the number of different locations in the cache where a particular block can be loaded. In an N -way set-associative cache, a particular block can be loaded in N different cache locations. Direct-mapped caches have an associativity of one, and can load a particular block only in a single location. Fully associative caches are at the other extreme and can load blocks anywhere in the cache.

High cache hit ratios depend on a program's stream of memory references exhibiting locality. A program exhibits temporal locality if there is a good chance that an accessed data item are accessed again in the near future. A program exhibits spatial locality if there is good chance that subsequently accessed data items are located closely together in memory.

2.3 Caches and Algorithm analysis

Since the introduction of caches, miss penalties have been increasing steadily relative to cycle times and have grown to the point where good performance cannot be achieved without good cache performance[5]. Unfortunately, many fundamental algorithms were developed without considering caching. Worse still, most new algorithms being written do not take cache performance into account. Despite the complexity that caching adds to the programming and performance models, cache miss penalties have grown to the point that algorithm designers can no longer ignore the interaction between caches and algorithms.

Lamarca[5] has demonstrated the potential performance gains of cache-conscious design in his dissertation. The performance results he obtained demonstrate that memory optimizations significantly reduce cache misses and improve overall performance.

A drawback of designing algorithms for cache performance is that often none of the cache parameters are available to the programmer. This raises a dilemma. A programmer might know that it is more efficient to process the data in cache size blocks but cannot do so when the capacity of the cache is unknown. One approach used by some is to make a conservative assumption and rely on the cache to be some minimum size. We take an

approach taken by Lamarca[5], and assume that the exact cache parameters are exported to the programmer by the system. That is, we assume that the capacity, block size, associativity and miss penalty of the caches are known by the programmer.

This change clearly increases the complexity of the programmer's environment. Caches, that traditionally were transparent to the programmer, are now exposed. This change also raises portability issues. While correctness still is preserved, codes compiled for one memory system might perform poorly if executed on a machine with a different memory system.

Despite these drawbacks, exporting cache parameters has the potential to aid efficient algorithm design greatly. Lamarca[5] showed in his thesis that efficient algorithms can be made to perform even better when specific architectural characteristics are known.

The majority of researchers in the algorithm analysis community compare algorithm performance using analyses in a unit-cost model. The RAM model, discussed in Cormen[1] and shown also in figure 1 is used most commonly, and in this abstract architecture all basic operations including reads and writes to memory, have unit cost. Unit-cost models have the advantage that they are simple to understand, easy to use and produce results that are easily compared. A serious drawback is that unit-cost models do not adequately represent the cost of memory hierarchies present in modern computers shown in figure 2. In the past, they may have been fair indicators of performance, but that is no longer true.

It is also common for the analyses of algorithms in a specific area only to count particular expensive operations. Analyses of searching algorithms, for example, typically count only the number of comparisons performed. The motivation behind counting only expensive operations is a sound one. It allows the analyses to be simplified yet retain accuracy since the bulk of the costs are captured. The problem with this approach is that shifts in technology can render the expensive operations inexpensive and vice versa. In this thesis a memory reference is considered to be the most expensive operation. Thus throughout this thesis, the theoretical complexity is assumed to be the number of memory references (instead of the number of comparisons).

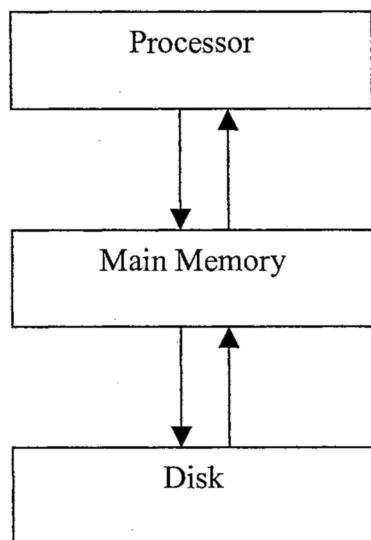


Figure 1. RAM Model

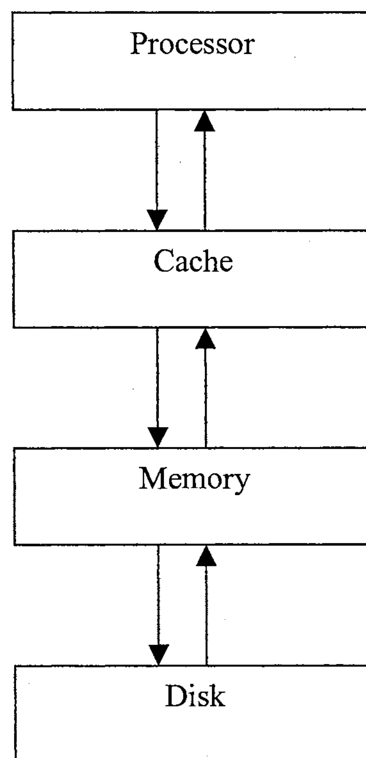


Figure 2. Memory hierarchy of Modern computers

Chapter 3

Cache Based Analysis of Insertion Sort

3.1 Insertion Sort

Insertion sort is a simple, stable, internal sorting algorithm which sorts N elements in $O(N^2)$ time in the worst case. It only is useful for sorting a small number of data items, but due to its simplicity and due to the fact that it sorts "in place", it is used in various linear sorting algorithms; such as, radix sort and bucket sort, to sort the intermediate buckets [1]. It is also used to sort small arrays in mergesort and quicksort [1]. The basic idea of insertion sort is also used in shellsort [10], though in a modified manner. Thus, its analysis in terms of cache miss rate can give valuable insight into other complex algorithms based on it or using it. Also, since it uses only constant amount of space outside of the original array it is very efficient in terms of memory space utilization.

Insertion sort is mainly useful in sorting a relatively small number of data elements, therefore the study of its cache miss rate is very important. Even a small number of misses cause large miss overhead since the sorted set is small. Therefore we predict the cache misses exactly, since even a small number of misses can affect insertion sort's performance drastically.

3.2 Cache based Analysis of Insertion sort

The first look at the algorithm gives an impression that it has good data locality of reference since it accesses the adjacent data element in the following access, which is an indication of good spatial locality.

3.2.1 Insertion Sort Source Code

```
    void sort ( double A[], int n )
    {
        double temp;
        int i, j;
     for ( i = 1; i < n; i++ )
     {
1.        temp = A[i]; // 1 reference
2.        j = i - 1;
3.       while ( j >= 0 && A[j] > temp ) // 1 reference
         {
4.         A[j+1] = A[j]; // 2 references
5.          j--;
         }
6.       A[j+1] = temp; // 1 reference
    }
```

Figure 3.

We are not concerned about the internal variables used by the algorithm. This is because they can be represented in registers and never be written to any part of memory. This, in

effect, means that the internal variables do not have a memory address and thus do not have any memory references. The main concern is the array elements we are trying to sort.

As shown in the algorithm above we have the following memory references:

1. Line # 1 : 1 reference
2. Line # 3: 1 reference
3. Line # 4 : 2 references
4. line # 6 : 1 reference

Our first task is to determine the exact number of memory references the algorithm makes to sort a given set of numbers.

Let us consider the following data cache model for the rest of the analysis:

Replacement Algorithm: **LRU (Least Recently Used)**

Cache Type : **Fully associative**

Cache size (number of elements cache can hold): **C**

Block size (number of elements in one block): **B**

Number of blocks in cache: **R**

Total number of Data elements to sort: **N**

Note: On a miss the entire block containing the missed element is brought into the cache.

3.2.2 Best Case Analysis:

Insertion sort has its best performance when the array is sorted already and it runs in O(N) or linear time. The number of references is 3N-3, from passes P=2 to N. The number of moves per element is zero, thus we just get only one compulsory miss per block. This fact is shown in figure 4, in which a block size B=4, is assumed. It is assumed, without loss of generality, that the array elements are same as their array index since this gives a simpler view of the sorted array. Due to this, the actual data elements have not been shown separately from the array indices.

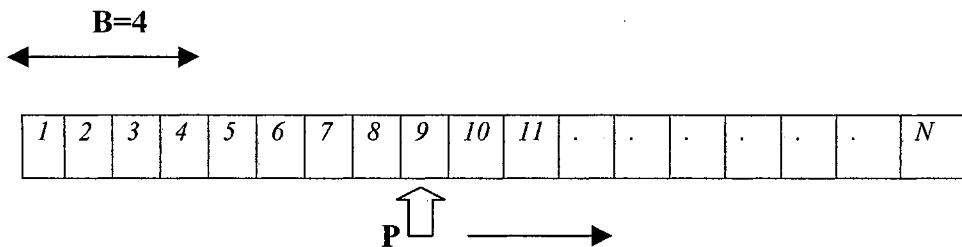


Figure 4.

We start the passes from P=2, which causes a compulsory miss in element 2. Due to this miss the whole block; i.e., elements 1 through 4 are brought into the cache, and we don't get a miss until the 5th element. Thus, we get misses every B elements accessed until the Nth element giving a total of $\lceil N/B \rceil$ compulsory misses. Therefore the miss rate is approximately 1/3B.

3.2.3 Worst Case Analysis:

The worst case performance of insertion sort $O(N^2)$ is very poor compared to its best case $O(N)$. Thus we concentrate heavily on its worst case cache performance since it would give a true idea of its data locality.

Given the above cache model the worst case analysis is based on the fact that the array is reverse sorted, therefore each element must be brought back to index 1 from its current position in the array at each pass. This means that in pass number P we must go back P-1 indices in the array to reach the first element.

This gives the number of references per pass as:

1 + 3*(P-1) +1**..(1)**

Referring to figure 5, the first term in the above equation represents the memory reference on line 1. The second term is the sum of all memory references on lines 3 and 4 over (P-1) moves. Finally, the last term is the memory reference on line 6.

The total number of references is the summation of (1) over all the N-1 passes (from P=2 to P=N).

Thus we get the total number of references

$$= \sum_{P=2}^{N} [2 + 3 * (P - 1)]$$

= 2*(N-1) + 3*[N*(N+1)/2 –N]..(2)

In order to find the total number of misses following two cases are considered:

Case I : N<=C

This is trivial since all the elements are in the cache there are only $\lceil N/B \rceil$ compulsory misses which is same as the best case.

Case II : N>C

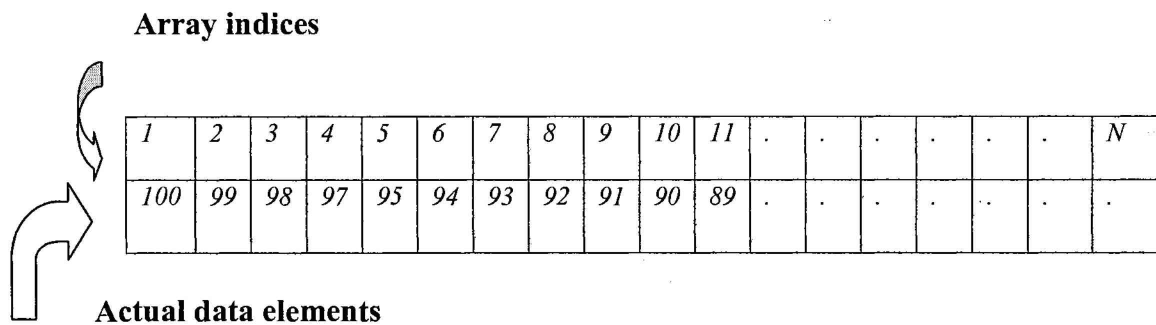


Figure 5.

Figure 5 shows a view of the reverse sorted array. For further analysis in this case only the indices of the array are given and it will be assumed that the data contained in them is in reverse sorted order.

Until the pass P= C, all passes cause only a total of $\lceil C/B \rceil$ compulsory misses since all the C elements can be stored in the cache.

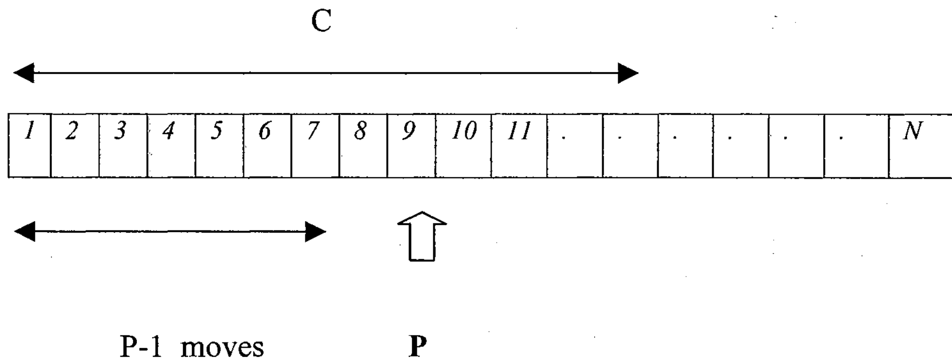


Figure 6.

For all passes P>C the roll-in/roll-out of blocks leads to capacity misses.

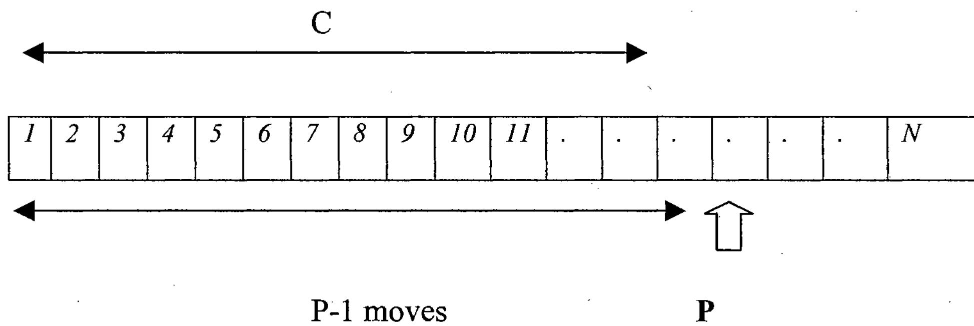


Figure 7.

In the Pth pass we try to place the Pth element into the correct position in the array and in the worst case it is placed in the first position.

This current element is referenced only once in that pass and then is copied into the ***temp*** *variable (which is internally a register) on line 1. Thus by the time we reference the first*

element in the array, the current block; i.e. the block containing the Pth element, becomes the LRU block and is replaced.

This is a very undesirable replacement since all elements after P>C lead to misses, because they are in LRU blocks and are replaced continuously. These misses are either capacity misses due to replacement of the current block or compulsory misses. Thus all the **N-C** elements that are sorted in the passes C <P < N cause misses since they will have already been replaced.

Also all the blocks before the current block cause misses since they are replaced just before they are accessed again; i.e., there are $\lfloor P/B \rfloor$ misses for the Pth pass for all previous (i.e. blocks to the left of the current block) blocks. The total number of capacity misses in previous blocks is the summation over all passes from C to N.

Thus total misses in the previous blocks are

$$\sum_{P=C+1}^{N} \lfloor (P-1)/B \rfloor$$

Total misses are the sum of all the misses discussed above,

$$\lceil C/B \rceil + N - C + \sum_{P=C+1}^{N} \lfloor (P-1)/B \rfloor \quad \textbf{(3)}$$

The first term is the compulsory misses until the pass P=C since the Pth element moves (P-1) indices back and all the accessed elements can be stored in the cache.

The second term indicates the fact that after P=C every element causes a miss.

The third term is for misses in the blocks to the left of the current block.

Contrary to one's first impression about locality, the cascade affect of replacing the current block destroys the locality of reference of the algorithm, and it keeps generating cache misses from passes P=C through and including P= N.

Note that if both C and B are chosen to be powers of 2, as has been done by LaMarca [5,6,7], the first term reduces to C/B and equation (3) reduces to

$$N-(C/B)(B-1)+\sum_{P=C+1}^{N}\lfloor (P-1)/B \rfloor \qquad \textbf{(3a)}$$

If N>>C then the miss rate is approximately 1/3B.

3.2.4 Average Case analysis:

The average case can be analyzed in a way similar to the worst case analysis except that the number of moves for the Pth pass is (P-1)/3 instead of P-1 as in the worst case. This means that we expect an element to move one-third the way back in the array on an average.

Thus the total number of references per pass is

1+3*(P-1)/3 +1..**(4)**

and the total number of references in the average case are:

$$= \sum_{P=2}^{N}[2+(P-1)]$$

=2*(N-1) + [N*(N+1)/2 –N]..(5)

In order to find the total number of misses following two cases are considered

Case I : N<=3C

Since each element only moves one-third way back we get just $\lceil N/(3*B) \rceil$ compulsory misses.

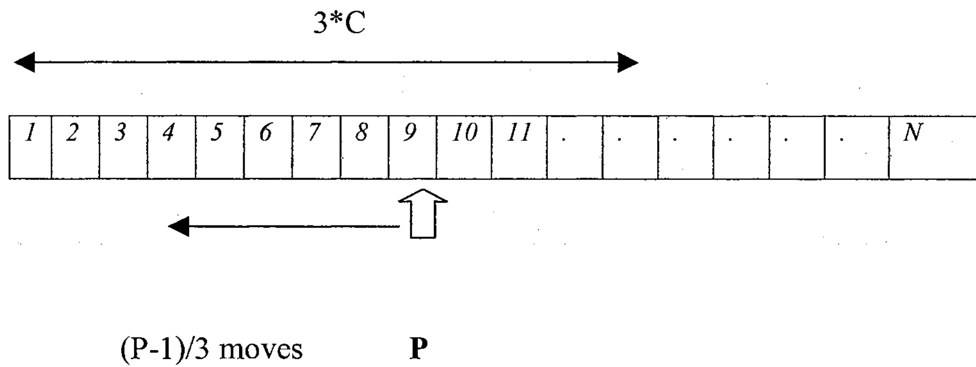


Figure 8.

Case II : N>3C

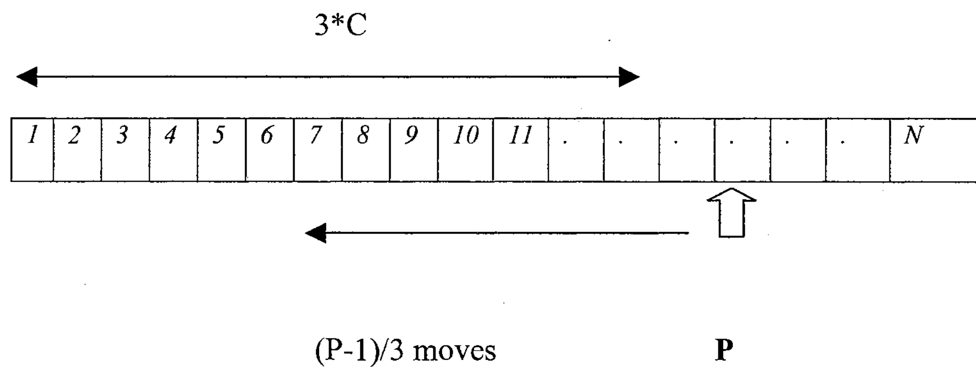


Figure 9.

With the same argument as the worst case analysis the total number of misses in the average case are expected to be:

$$\lceil 3*C/B \rceil + N - 3*C + \sum_{P=3*C+1}^{N} \lfloor (P-1)/(3*B) \rfloor \text{..} \textbf{(6)}$$

The first term is the compulsory misses until the P=3*C pass is reached since it will move only one-third the distance back, and all the accessed elements can be stored in the cache.

The second term indicates the fact that after P=3*C every element causes a miss.

The third term is for cache misses in the blocks to the left of the current block.

Again if both C and B are chosen to be powers of 2, the first term reduces to 3*C/B and equation (6) reduces to

$$N - 3*C/B(B-1) + \sum_{P=3*C+1}^{N} \lfloor (P-1)/(3*B) \rfloor \text{..} \textbf{(6a)}$$

As in the best and worst cases described above if N>>C then the miss rate is again 1/3B approximately.

Chapter 4

Simulation and Comparative Graphs

4.1 Simulation

A simulation program was written to analyze the cache performance of insertion sort. The listing of the program is given in the Appendix.

The general idea of the simulation is given below:

- The Simulation checks all the instructions that reference the array indices; i.e., lines 1,3,4 and 6, incrementing the number of references for each pass.
- Each reference is time stamped so that the most recent reference time is recorded.
- For each of these references, the simulation it checks if they are a hit or miss.
- On a miss the number of misses are incremented and the whole block containing the missed index is brought into cache and time stamped.
- LRU is used for block replacement. This is done by checking the time stamps of the blocks and finding the block with the oldest time stamp.

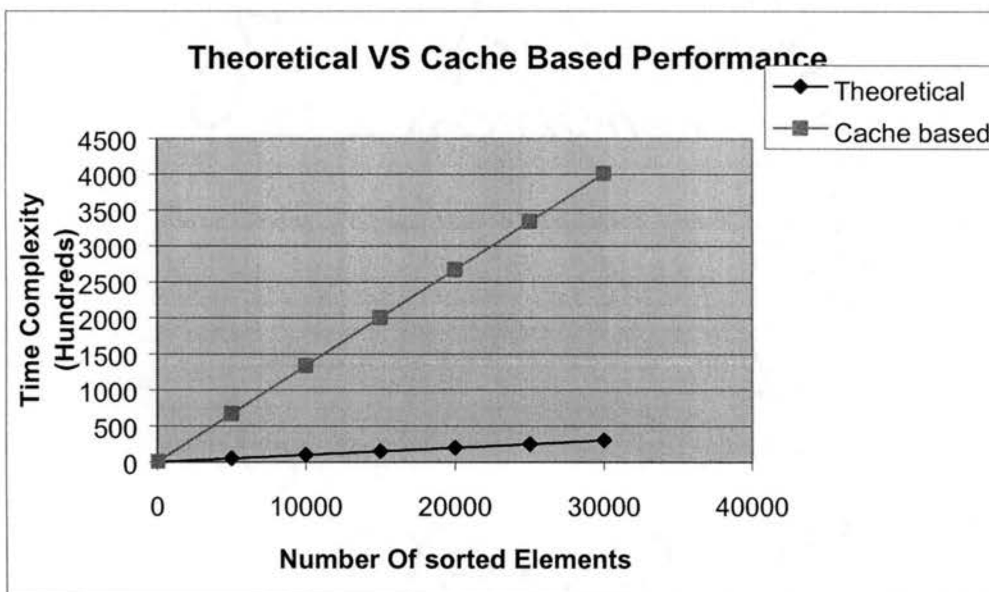
4.2 Graphs

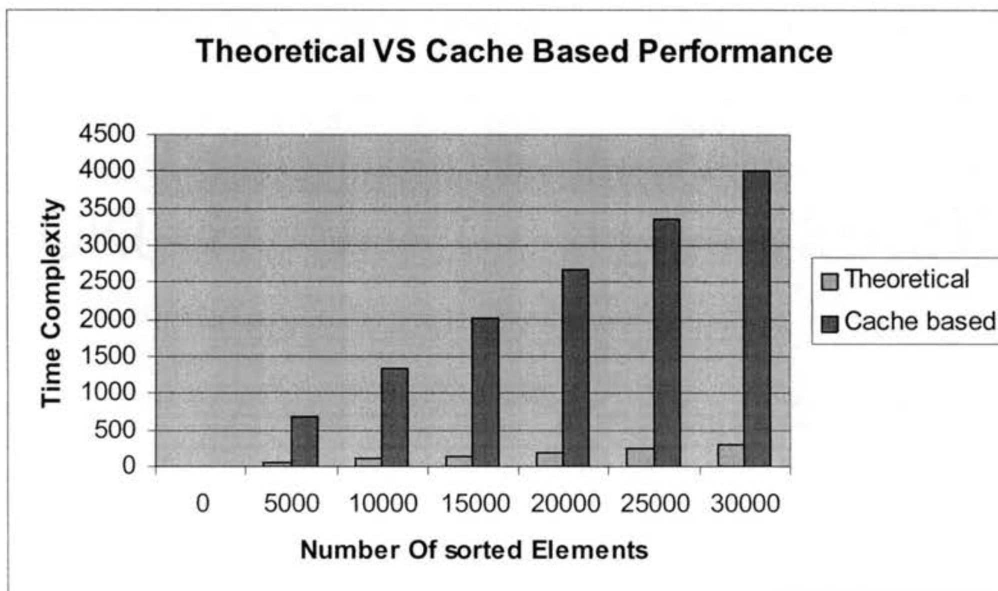
All the graphs assume the following cache parameters:

Cache size C = 512 words, Block size B= 8 words, Miss penalty = 100 cycles (According to Lamarca[5] cache miss penalty is 100-120 cycles.)

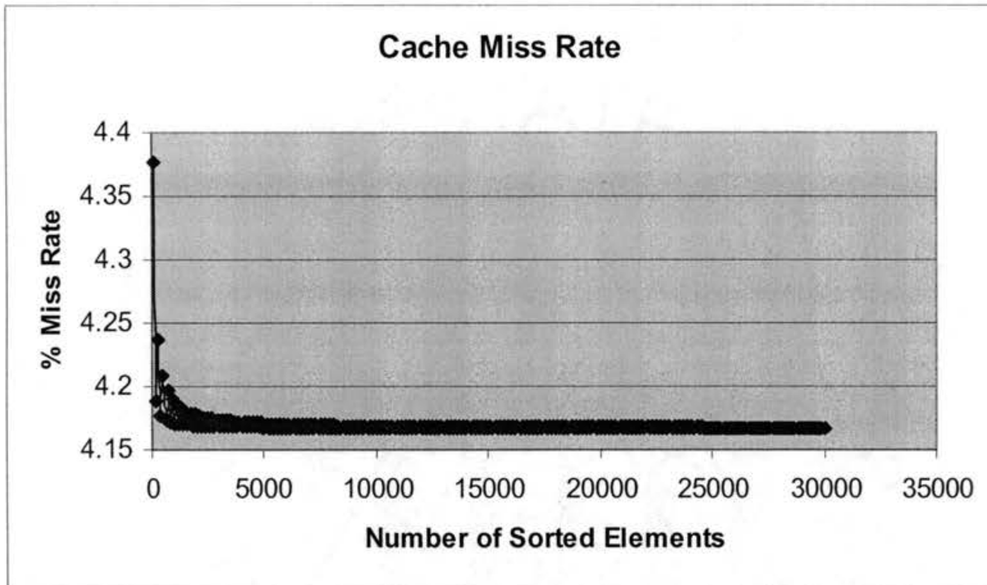
4.2.1 Best Case Graphs

1. Cache Based VS Theoretical Time (# of memory references) Complexity

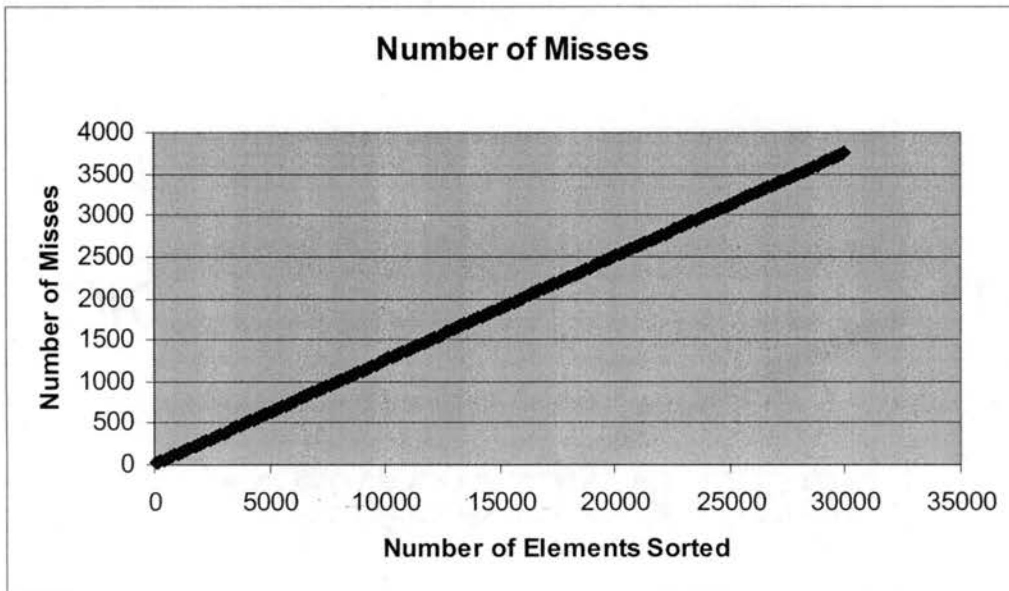




2. Cache Miss Rate

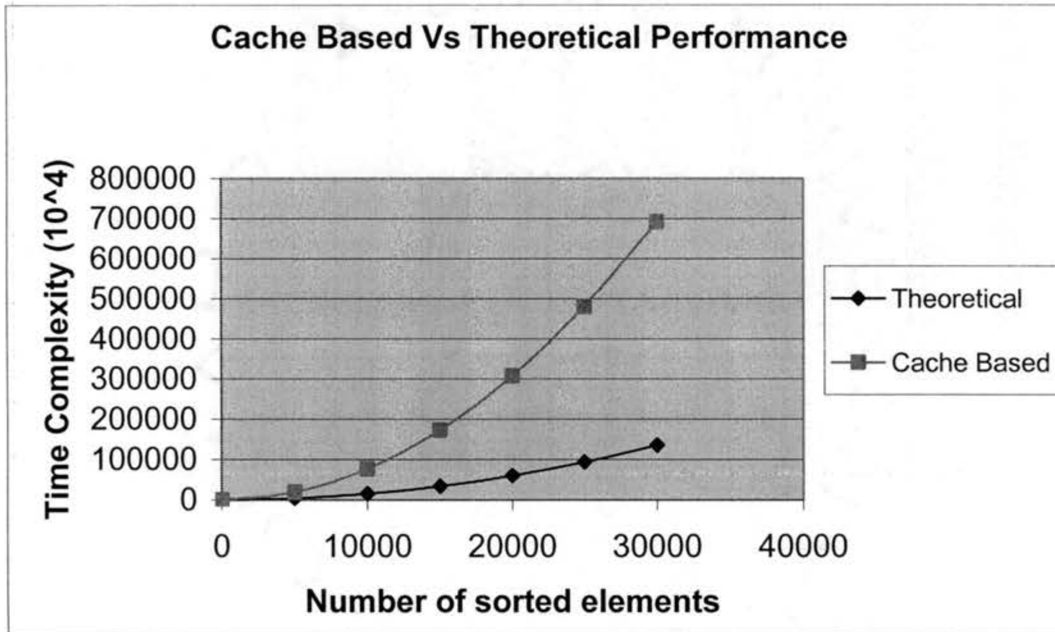


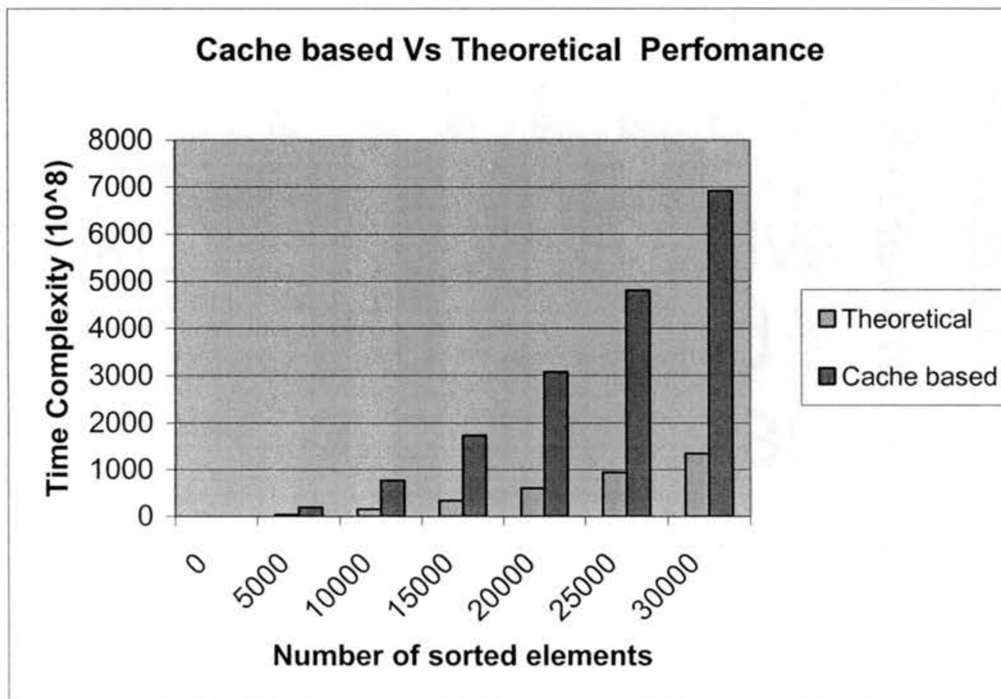
3. Number Of Cache Misses



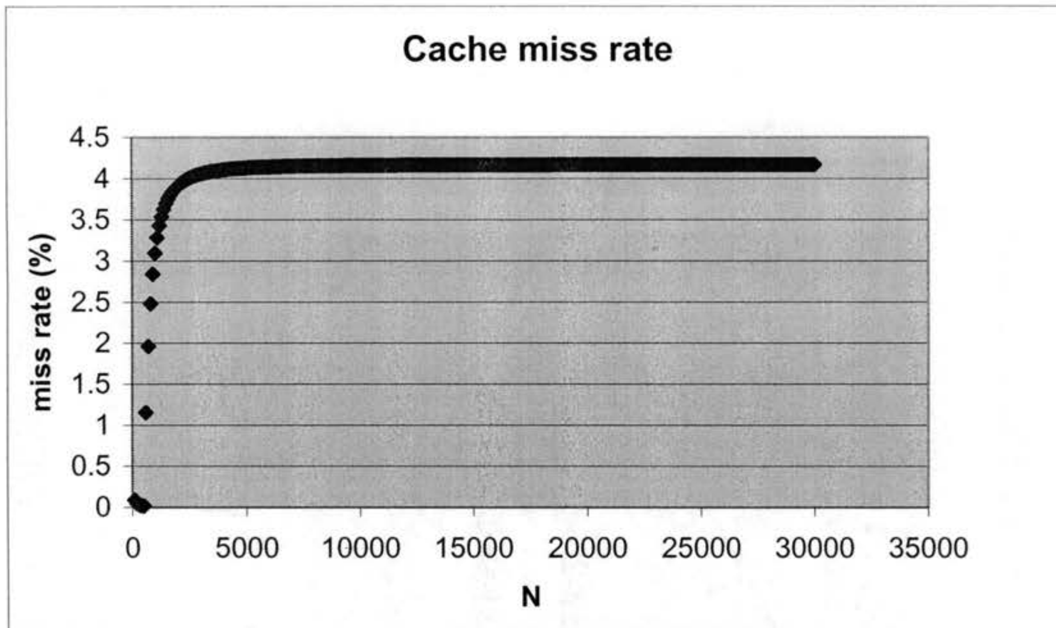
4.2.2 Worst case

1. Cache Based VS Theoretical Time Complexity

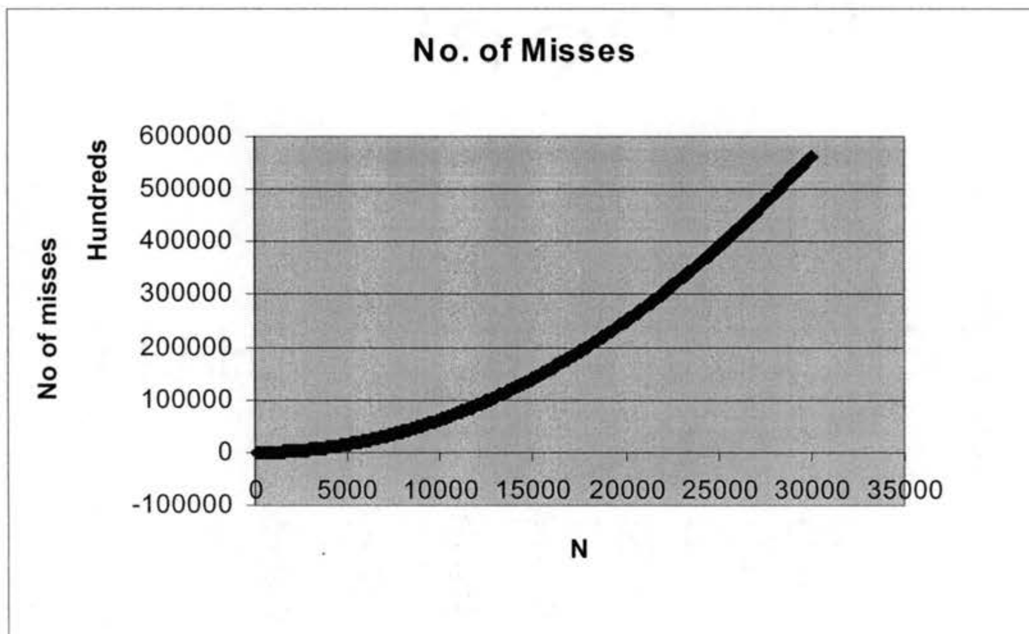




2. Cache Miss Rate

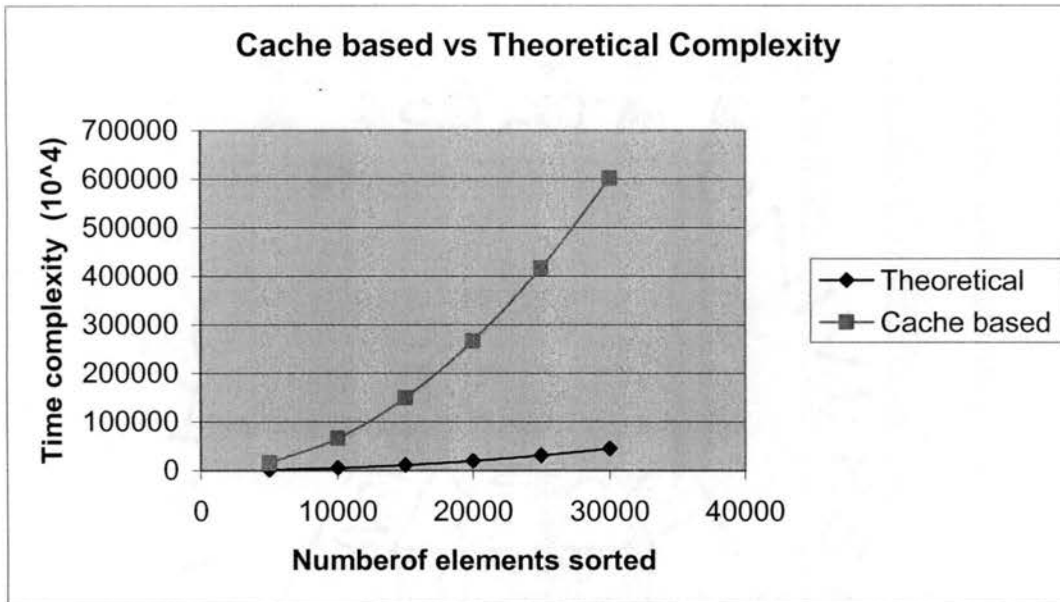


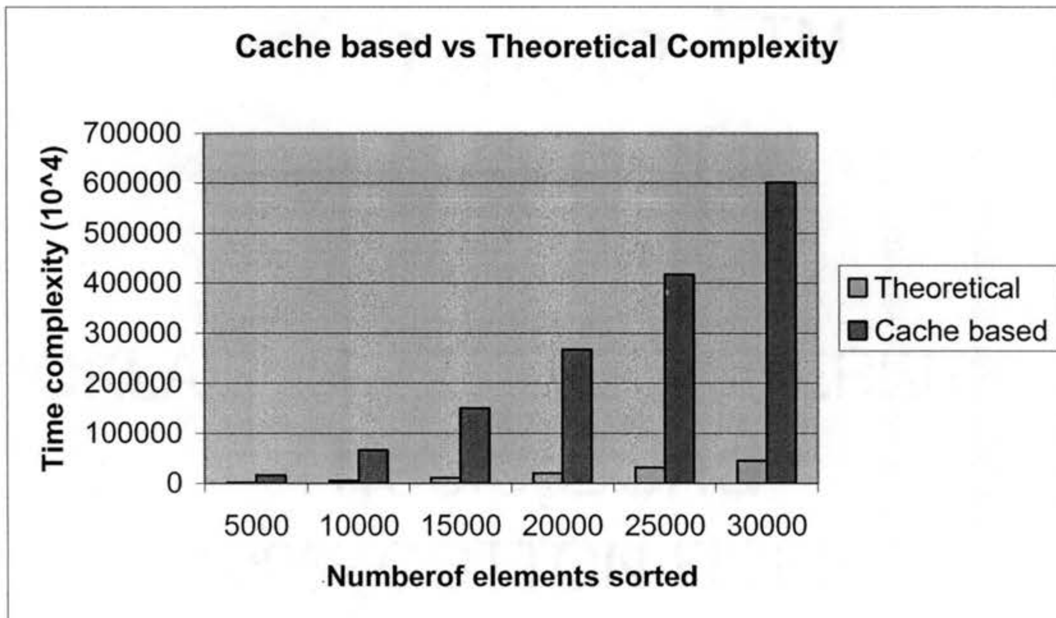
3. Number Of Cache Misses



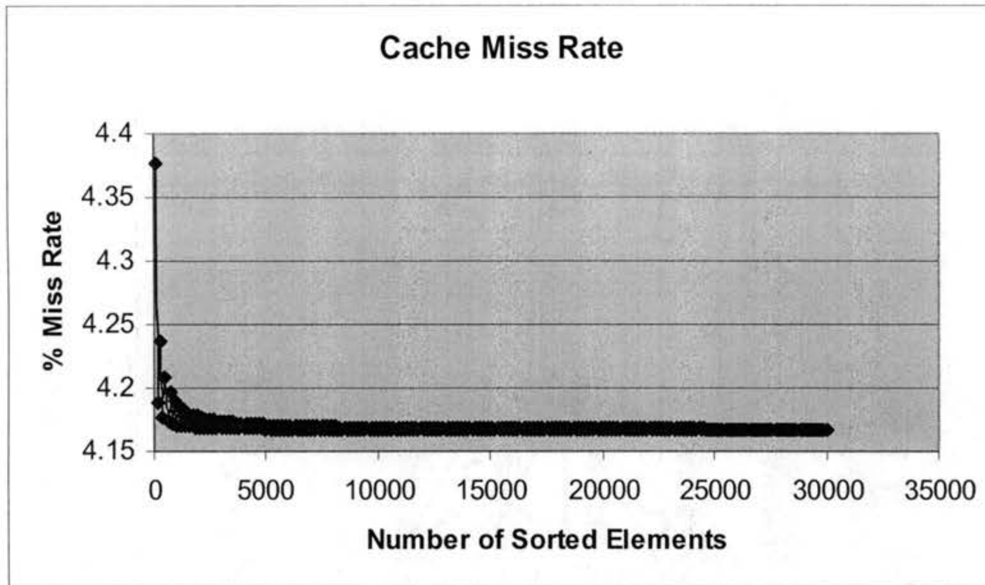
4.2.3. Average case Graphs

1. Cache Based VS Theoretical Time Complexity

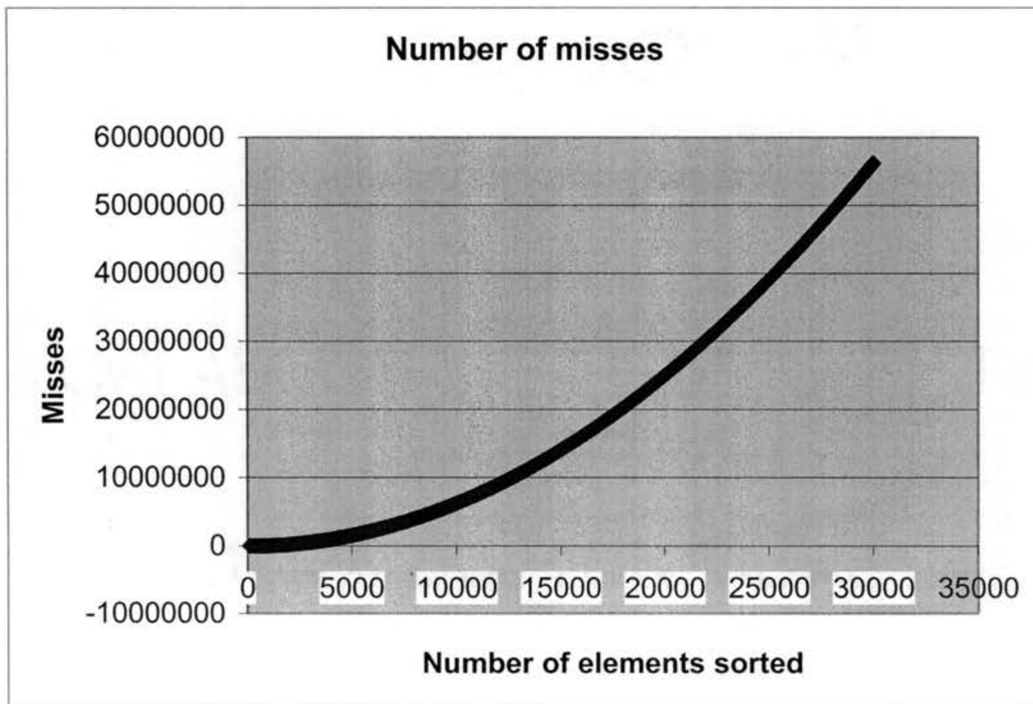




2. Cache Miss Rate



3.Number of Cache Misses



5. Observations:

The following observations can made from the above graphs:

- There is a remarkable difference between the cached and non-cached performance of insertion sort and this difference increases as the number of sorted element's increases.
- The miss rate saturates to a constant level (=1/3b approx.) and is inversely proportional to the cache block size as the number of elements sorted increase.
- The number of misses increase linearly in the best case while both in the Average and Worst case it is a quadratic curve.

Chapter 5

Conclusion and Future Work

The main aim of this thesis is to analyze the cache based performance of insertion sort and compare its cache based performance with its theoretical complexity. Closed form solutions are developed for the number of cache misses and verified using simulations. Finally, comparative graphs are plotted to visualize the behavior of insertion sort in the presence of caches.

From the above analysis and graphs, it can be concluded that cache heavily affects the performance of insertion sort. The main conclusion that can be drawn from this work is that the influence of caching needs to be taken into account in the design and analysis of algorithms.

Future work in this area can be divided broadly into two parts. First is to make modifications to insertion sort in order to improve its cache behavior. This would involve locating the parts of the algorithm that cause the bulk of cache misses and improving on them. Second is to extend the above analysis to various other algorithms. These algorithms could either be ones that are based on insertion sort, such as shellsort, or variants of algorithms, like quicksort and mergesort, which use insertion sort.

Bibliography

[1] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Riverst, "Introduction to Algorithms", MIT Press, MA and McGraw-Hill Book company, NY, 1998.

[2] Hennesey, J. and D. Patterson, "Computer Architecture A Quantitative Approach". Morgan Kaufman Publishers, Inc., San Mateo, CA, 1996.

[3] Knuth, D., "The Art of Computer Programming: Volume 3: Sorting and Searching", Addison –Wesley, Third Edition, Reading, MA, 1998.

[4] Lam, Monica S., E. Rothberg, and M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), (Santa Clara, CA, 8-11 April 1991). SIGPLAN Notices, Volume 26, Number 1, pp. 63-74, 1991.

[5] LaMarca, A., "Caches and Algorithms", Ph.D. Dissertation, University of Washington, Seattle, 1996.

[6] LaMarca A. and R.E. Ladner, "The Influence of Caches on the Performance of Sorting", Journal of Algorithms Volume 31, pp. 66-104, 1999.

[7] LaMarca A. and R.E. Ladner, "The Influence of caches on performance of heaps", ACM Journal of Experimental Algorithmics (on-line journal), Volume 1, Number 4, 1996.

[8] Nyberg, Chris, Tom Barclay, Zarka Cvetanovic, Jim Gray, Dave Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort", VLDB Journal, Volume 4, Number 4, pp. 603-627, 1995.

[9] Panda, Preeti R., Hiroshi Nakamura, Nikil D. Dutt, Alexandru Nicolau, "Augmenting Loop Tiling with Data Alignment for Improved Cache Performance", IEEE Transactions on Computers, Volume 48, Number 2, pp.142-149, February 1999.

[10] Sedgewick, Robert, "Analysis of Shellsort and Related Algorithms", Proceedings of the Fourth Annual European Symposium on Algorithms - ESA'96", Lecture Notes in Computer Science 1136, Barcelona, Spain, September 1996.

[11] Shi, Ying and Eushiun Tran, "Performance Analysis of Sorting Algorithms", Term Paper for Advanced Computer Architecture, Carnegie Mellon University Pittsburgh, Pennsylvania, March 1999.

[12] Weiss, Mark A., "Data Structures and Algorithm Analysis in C", Second Edition, Addison-Wesley, 1997.

Appendices

APPENDIX A

Glossary of Terms

Cache: In order to speed up memory accesses, small high speed memories called caches are placed between the processor and the main memory. Accessing the cache is typically much faster than accessing main memory

Fully Associative Cache: This type of cache can load blocks anywhere in the cache.

Internal Sorting: Internal Sorting means that the entire data structure to be sorted can be held in the computer's main memory [12].

In-place Sorting: In-place means that the amount of storage space we need for our data during the execution of the algorithm is constant [1].

Least Recently Used: A Replacement algorithm in which the cache block that was least recently used is replaced.

Miss rate: This is the ratio of the number of cache misses to the total number of memory references.

Stable Sorting: A sorting algorithm is stable if the elements with equal keys are left in the same order as they occur in the input [1].

APPENDIX B

Simulation Program Source Code

```
/*****************************************************************
Simulation Program  for analyzing the Cache Based Performance of Insertion Sort and
compare the simulated results with the ones Developed theoretically .
This simulation compares the theoretical results obtained for the worst case performance
of insertion sort with the simulated results of the same. It was observed that the
theoretically obtained closed form solutions exactly matched the simulated results.

*****************************************************************/

#include <fstream.h>
#include <string.h>
#include<iomanip.h>
#include<stdio.h>
#include<math.h>
#include <ctype.h>



// Total number of elements
const long N =2000;
// block size
const long B=4 ;
// # of blocks
const long R=128 ;

//C=B*R = 512

long temp , A[N];
long last_ref[R] ;
long cache[R][B];
long clock =0 ;
long miss_no=0 ;

/**********************************************************
*********
  This funtion returns if the reference passed to it was a
hit or miss

**********************************************************
*********/
```

```
bool miss(long ref)
{

     //check all blocks to find ref

   for(int i=0 ; i<R ; i++)
   {
    if(cache[i][0]==B*int (ref/B)) // check first element
in block
     return false  ; //hit  , assumes that the ref is ther
if one elem of its block is ther
   }
   return true ; // miss
}
/***********************************************************
*********
  This function returns the Block number of the address
passed to it

************************************************************
*********/
long get_block(int address)
{
     for(int r=0 ; r<R ; r++)
     {
           for(int b = 0 ; b < B ; b++)
           {
                if(cache[r][b] ==address)
                     return r ; // find block
           }
      }
}
/***********************************************************
*********
  This  function returns the LRU block
************************************************************
*********/
long LRU()
{
 long  min = last_ref[0]  ;
 long lru =0  ; // block num of lru
     for(int i =1  ; i<R ; i++)
     {
          if(last_ref[i] < min)
          {
               lru =i ;
               min=last_ref[i] ;
```

```
        }

    }
    return lru  ;
}
/**********************************************************
  This Function inserts the array index in cache , actually
a full block

**********************************************************/

void insert(int address, long & block_no)
{
    block_no =LRU() ; // returns lru block
   //only putting address since just a simulation
     int add =B* int(address/B) ; // first address of the
block in array

   for(int i=0 ; i< B ; i++) // put whole block in
   {
       cache[block_no][i] = add+i ; // putting in all
address in the block in the cache

   }



}

void  main ()
{


      for(int r=0 ; r<R ; r++)
      {  last_ref[r] = -1 ;
           for(int b = 0 ; b < B ; b++)
           {
                cache[r][b] = -1 ; // empty cache
           }
      }



     long k , block_no , ref=0;
      // put distinct numbers in reverse order for worst
case
     for(k=0 ; k<N ; k++)
```

```
          A[N-k-1] = k ;

    int i, j;
    /*********Insertion Sort*************/

    for ( i = 1; i <N ; i++ )
     {
          temp = A[i];// 1 refrence
         ref++ ;
          if(miss(i))
          {
               insert(i ,block_no) ; // this will insert
whole block of sixe B
               last_ref[block_no] = clock ; //put in the
reference
               miss_no++ ;

          }
          else
         last_ref[get_block(i)] = clock ; //put in the
reference

          clock++ ;
         j = i - 1;

        while ( j >= 0 &&A[j] > temp ) //1 refernce
        {
        /********************/
            ref++ ;
          if(miss(j))
          {
               insert(j, block_no) ; // this will insert
whole block of size B
               last_ref[block_no] = clock ; //put in the
reference
               miss_no++ ;

          }
          else
        last_ref[get_block(j)] = clock ; //put in the
reference
         clock++ ;
          /********************/


         A[j+1] = A[j]; // 2 references
```

```
          /*********************/
          ref+=2 ;
          if(miss(j+1))
          {
               insert(j+1, block_no) ; // this will insert
whole block of sixe B
               last_ref[block_no] = clock ; //put in the
reference
               miss_no++ ;

          }
          else
          last_ref[get_block(j+1)] = clock ; //put in the
reference
          clock++ ;
          /*********************/


          if(miss(j))
          {
               insert(j , block_no) ; // this will insert
whole block of sixe B
               last_ref[block_no] = clock ; //put in the
reference
               miss_no++ ;

          }
          else
         last_ref[get_block(j)] = clock ; //put in the
reference
          clock++ ;
          /*********************/
         j--;
       }
       A[j+1] = temp; // 1 reference  same as above
       /*********************/
       ref++ ;
      if(miss(j+1))
          {
               insert(j+1 , block_no) ; // this will insert
whole block of sixe B
               last_ref[block_no] = clock ; //put in the
reference
               miss_no++ ;
```

```
            }
        else
        last_ref[get_block(j+1)] = clock ; //put in the
reference
        clock++ ;
        /*********************/

    }


      /*********Insertion Sort Code Ends here************/

       // Results

      long C=B*R  , nc= long(N-C >0 ? N-C :0) , ncb =
ceil(nc/B);
      double REF = 2*(N-1) + 3*(N*(N+1)/2 - (N)) ;
    if(N>C)
    {
       long x= (N>C ? (1.0*C)/B : (1.0*N)/B ) ;
      long y = x + (N-C) ; // const part when N>C
       cout<<"\n Const Part = " << y << endl ;
       long sum = 0 ;
       for(int p=C ; p<N ;p++)
       {
            sum+=int(p/B) ;

       }
     cout<<"\n Expected # of refs   :"<<REF ;
      cout<<"\n Expected number of misses  :"<< y+sum ;
   cout<<"\n Expected miss rate  :"<<100*(1.0*(y+sum))/REF ;
    }

   else
   {
      cout<<"\n Expected # of refs   :"<<REF ;
     cout<<"\n Expected number of misses  :"<<
     ceil((1.0*N)/B) ;
     cout<<"\n Expected miss rate  :"<<  100*
     double(ceil((1.0*N)/B)) /REF ;  //(N*N)  ;
   }



cout<<"\n\n Actual # of refs   :"<<ref ;
```

```
cout<<"\n Actual   number of misses  :"<< miss_no ;
cout<<"\n Actual miss rate  :"<<100*
(miss_no*1.0)/ref<<endl ;

}
```

VITA

SUNIL MEHTA

Candidate for the Degree of

Master of Science

Thesis: INFLUENCE OF CACHES ON THE PERFORMANCE OF INSERTION SORT

Major Field: Computer Science

Biographical:

Education: Graduated from AEJC High School, Bombay, India in June 1994; received Bachelor of Engineering degree in Mechanical Engineering from University of Bombay, Bombay, India in May, 1998. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2003.

Experience: Software Engineer, Mahindra British Telecom, Bombay, India, August, 1998 to July, 1999; Teaching Assistant, Department of Mechanical Engineering, Oklahoma State University, August, 1999 to May, 2000; Computer Scientist, Computer Aided Technology Transfer (CATT) Lab., Oklahoma State University, May, 2000 to May, 2002; Software Developer, Intern, Seagate Technology, Oklahoma City, May, 2002 to August, 2002; Computer Scientist, Computer Aided Technology Transfer (CATT) Lab., Oklahoma State University, August, 2002 to December 2002.