Volume : 1, Issue : 2
July - December 2011

ISSN : 2229 - 3515

AUTHOR'S PERSONAL COPY

International Journal of

ADVANCES IN SOFT COMPUTING TECHNOLOGY

Editor-in-Chief
Dr. Vaka Murali Mohan

Editor-in-Chief
BHAVAN RESEARCH CENTER

© 2011 BRC

Algorithms and Data Structures: Efficient and Cache-Oblivious

Ritika Angrish* and Dr. Deepak Garg

Department of Computer Science and Engineering, Thapar University, Patiala, Punjab, India

Keywords
Cache-oblivious Search trees, Cache-oblivious Hashing, String Sorting, Integer multiplication

ABSTRACT: *The Computer Architecture consists of memory hierarchy which varies from fast and expensive to cheap and slower. And with increasing speed of the processors the time taken to transfer data between memories is more than the actual time taken to process it. To alleviate this cache-oblivious algorithms and data structures are developed. In this paper we discuss various cache-oblivious data structures like B-tree and hash table implementing cache-oblivious hashing; and cache-oblivious algorithms like integer multiplication and string sorting with improvement.*

1. Introduction

When the first computer was invented the main drawback was the speed of the processor and we had only one main memory to store and retrieve data. During those times the program performance and efficiency depended upon the number of instructions it contains and memory references was not considered. The model then used for finding the complexity was the RAM-model [2]. With the development in technology the speed of processor increased following the Moore's Law stating the increase in transistors to be more than 55% every year [11]. In contrast to this the increase in speed of memory has been slow i.e., nearly 7% per year [9]. Thus implying that the speed of memory will be slower than the processor speed and this difference will increase over time.

Hence to handle this problem of mismatch in speed memory hierarchy was introduced. In this a fast and expensive memory is placed close to CPU and slow and cheaper memories are placed far from the processor. The memory closer to processor are expensive that is why they are small and thus holds less data i .e., acting as a buffer for slower memories.

The hierarchy is constructed considering the locality of reference, mainly temporal locality which is a memory location once referenced by a program is likely to be used again several times within short interval, and spatial locality which is once a program has referenced a memory location it is likely to refer the nearby locations shortly. The fast memory placed near the processor is generally called the cache. When a program reference a data which is not present in cache and has to be accessed from the next level of memory is called a cache miss.

*RITIKA ANGRISH
Department of Computer Science&Engineering
Thapar University, P.O.Box. 32,
Patiala – 147 004, Punjab, India
E_mail: rangrish@gmail.com
Phone: 91-175-2393007/3688

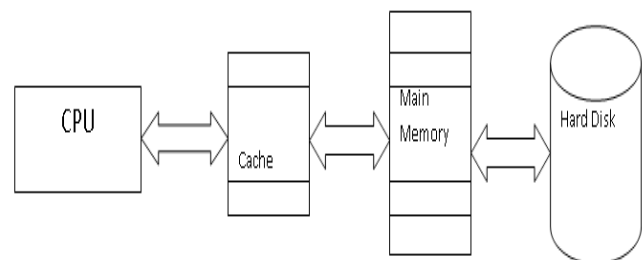


Fig. 1: Memory Hierarchy[i]

With the development in hardware, the cost of accessing different levels in hierarchy has created implications like algorithms processing large data are dependent on relative memory latency for them to have increased performance, and the performance of processor bound problems is also limited by the memory performance. To make the algorithms perform better the programmers should take advantage of the memory hierarchy.

2. Literature Review

A. *RAM Model*

Under RAM model the run time of algorithm is measured by counting the number of steps (instructions) an algorithm contains. It

assumes a flat memory system. It is the simplest model representing the working of computer. The main drawback of the RAM model was that it considered only one memory and is not an approximation of the actual hardware. This raised the need for new models to measure the performance and efficiency of algorithms correctly.

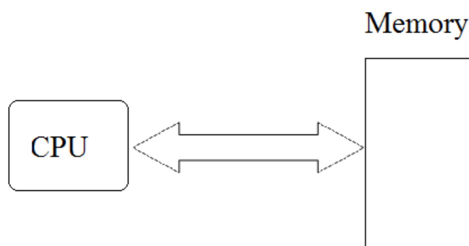


Fig. 2: The RAM-Model

B. *External Memory Model*

To overcome the drawback of RAM model an external memory model is considered. It is a two-level memory hierarchy model with block transfer also known as the I/O model, the disk access model or cache-aware model [1]. The basic idea was to compute the efficiency of algorithm on the basis of number of disk access as calculating number of instructions is not a solution in the case of memory hierarchy. The model contains a cache which is a fast memory placed near processor but its size is limited and a disk which is relatively slow memory placed at a distance from the processor having a limitless size.

In external memory model, the algorithms are specifically tuned according to the memory parameters, for example, in sorting [1] the optimal number of I/Os needed is □$(N/B \log_{M/B} N/B)$, where N is the number of elements to be processed, M is the number of elements that fit in internal memory and B is the number of elements that fit in one block of the internal memory. The limitation of this is that the implementation is highly platform independent as the size of memory varies from system to system and also the complexity of algorithm increases proportional to the increase in memory levels. In this model it is the responsibility of the programmer to control the block transfer. The external memory algorithms dependent on the memory parameters like block size or memory size are called external-memory algorithms or cache-aware algorithms.

C. *Cache-Oblivious Model*

The basis behind the cache-oblivious model is that designing external memory algorithms without knowledge of memory parameters. The cache-oblivious algorithm overcomes the limitations imposed to external memory algorithms i.e., they are not architecture independent and it is very difficult to adapt them to multiple levels of memory. A cache-oblivious algorithm as described by Prokop [8] is the one in which problem variable is independent of the memory parameters like cache size or block size; and is tuned to reduce the number of cache misses. Considering this definition the RAM model algorithms can also be considered as cache-oblivious. These algorithms are evaluated in an ideal-cache model.

The ideal-cache model consists of two-level of memory hierarchy i.e., a small cache and a very large main memory as in Fig. 3.

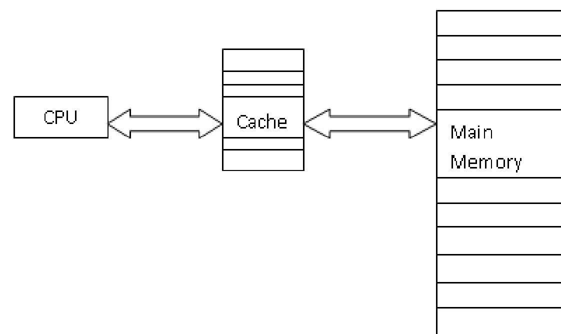


Fig. 3: The ideal-cache model[ii]

Table I: Comparison of Computational Models

Model / Parameters	RAM Model	External Memory Model	Cache-Oblivious Model
Complexity	Number of instructions	Number of disk access	Number of memory transfers
Model Type	Simple	Cache aware	Cache-Oblivious

Cache-oblivious data structure and algorithms are designed so that they can perform well at all levels of the memory hierarchy. It is seen that some cache-oblivious algorithms like matrix multiplication algorithm [14], cache-oblivious priority queues [4] when compared to their cache conscious counterpart were found superior.

3. Cache-Oblivious Data Structures

In this section we discuss about the various cache-oblivious data structures that have been proposed till date. This includes cache-oblivious trees both static and dynamic and hash table using cache-oblivious hashing.

A. Cache-oblivious B-Trees

B-Tree is a balanced tree data structure that keeps data stored and allows searches, sequential access, insertions and deletions in logarithmic amortized time. They are optimized tree structures used when part or the entire tree must be maintained in secondary storage. Cache-oblivious B-Trees are the one that perform well for all levels of the memory hierarchy and do not depend on the number of memory levels, the block size and number of blocks at each level.

i) Static B-Trees

The static cache-oblivious search tree having search cost equal to the search cost of standard cache aware B-tree (O ($\log_B N$) I/Os) was first proposed by Prokop [13]. The search tree is related to the van Emde Boas layout as shown in fig.4.

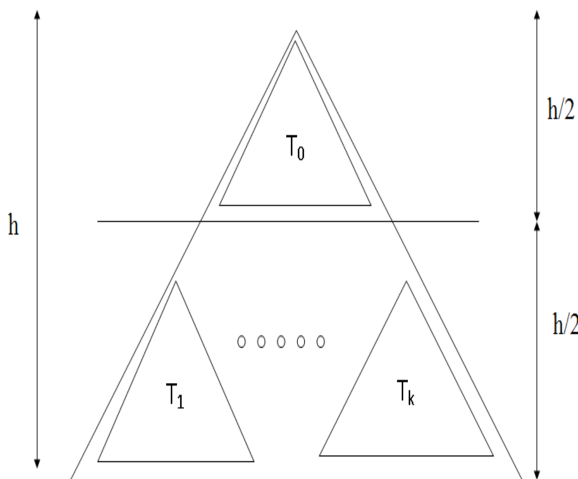
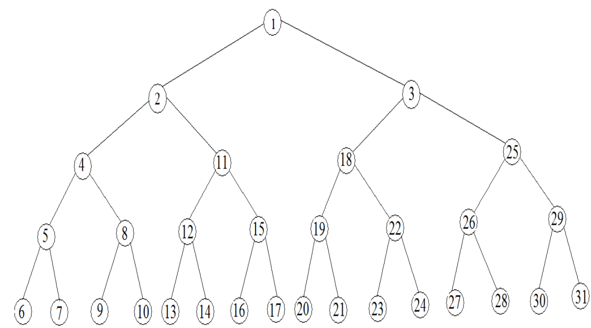


Fig. 4: The Van Emde Boas Layout

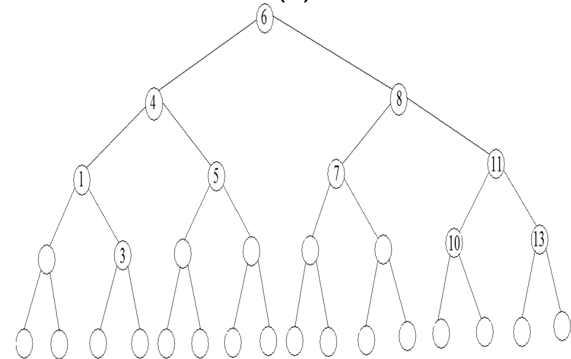
The van Emde Boas layout of a complete binary tree T is to have a recursive layout by dividing the tree at the middle level. In this a tree having a single node is said to be a trivial case and it is set in memory as the single node. In other words a tree of height h=logN, have the top tree T_0 as a sub tree consisting of nodes at the topmost $\lfloor h/2 \rfloor$ levels of T and the bottom $T_1,\ldots, T_k$ trees to be the □($\sqrt{N}$) sub trees rooted in the nodes on level $\lceil h/2 \rceil$ of T.

ii) Dynamic B-Trees

The basic ideaof dynamic B-Tree [5] was to embed a dynamic binary tree of height logN+O(1) onto a static complete binary tree which in turn is embedded into an array using the van Emde Boas recursive layout as shown in fig.5. Techniques for maintaining small height in a binary tree used to maintain the dynamic tree takes O($\log^2 N$) amortized time per update. The algorithm works by rebalancing the sub tree rooted at the node on whose leaf the violation occurred.



(a)



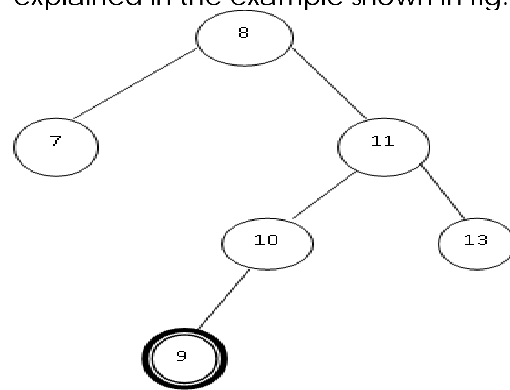
(b)

Fig. 5

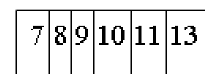
(a) van Emde Boas Layout of a complete binary tree of height 5.

(b) Embedding a search tree of height 4 on a complete binary tree of height 5[6]

The search procedure in the algorithm is same as the search procedure in binary tree but balancing needs to be done when we insert a node and an imbalance is created. This is explained in the example shown in fig.6.

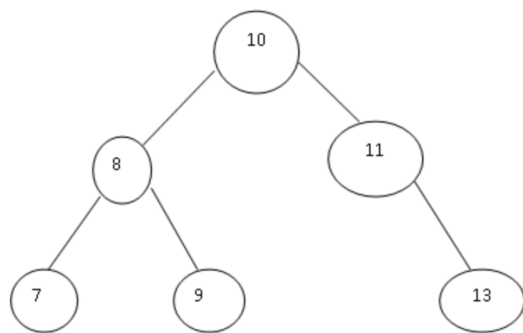


(a)



Middle Element

(b)



(c)

Fig. 6

(a) insertion of node 9 creates imbalance
(b) a sorted array of all elements of sub tree is created and a middle element if found
(c) the sub tree formed after rebalancing

The major advantages of a dynamic cache-oblivious B-tree are it minimize the number of memory transfers and no pointers are used which gives better space utilization as van Emde Boas layout is used which is an implicit layout.

B. Hash Table using Cache-oblivious hashing

Hash table is one of the simplest and most important index structures in databases. In this a hash function is used to map keys into their positions and various collision resolution strategies are used such as linear probing, chaining etc. The cache-oblivious hashing [12] uses linear probing as a collision resolution strategy ignoring the blocking (not considering the memory blocks) as chaining strategy would perform worse cache-obliviously because the list associated is not laid out consecutively. The search cost of cache-oblivious hashing was found to be $1+O(\alpha/b)$ I/Os where α is the load factor. Average search time of $1+2^{\Omega(b)}$ [10] can also be obtained in cache-oblivious hashing thus matching the cache-aware bound under the following two conditions: (a) b is a power of 2; and (b) every block starts at a memory address divisible by b.

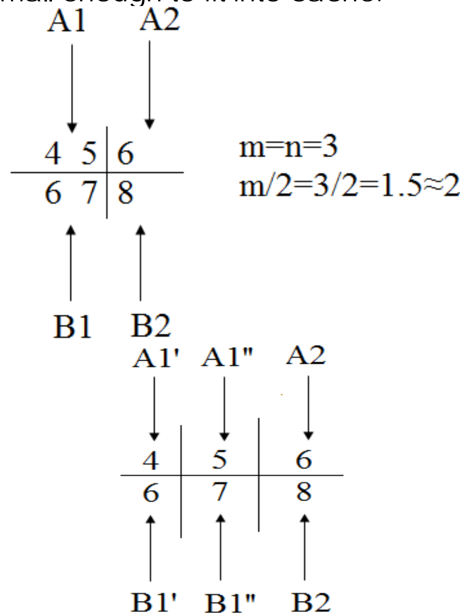
4. Cache-Oblivious Algorithms

A. Cache-oblivious Large Integer Multiplication

The large integer multiplication algorithm using divide-and-conquer [15] uses □(m^2) work and incurs □($m^2/LZ+m/L$) cache misses where m and n are the lengths of the two integers, m being the length of the largest integer, Z is the cache size and L is the line length of the cache. The concept behind this is to divide the two integers into two equal length parts and diving the sub problems further till m=1, in which case only two digits are multiplied and in the end combining the entire result. The algorithm is explained in the example.

Example: Multiply 456 with 678

1. Recursively divide the two numbers into equal parts till m=1or the sub problems are small enough to fit into cache.



2. Solve by multiplying A2×B2, A1"×B1", A1'×B1', A1"×B2 and so on , and then combine all the solutions to get the final result.

A2×B2= 8×6= 48

A1'×B2= 5×8= 40 append a 0 in the end= 400

A1"×B2=4×8=32 append two 0s in the end=3200

And so on the sum is equal to the multiplication of the numbers.

B. Improved Cache-oblivious string sorting Algorithm

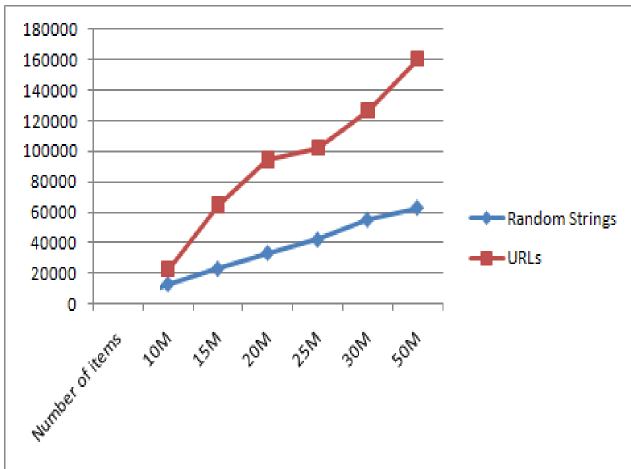
We present an improved cache-oblivious string sorting algorithm which uses concept of tried linear hashing to hash the strings to be sorted to a trie which in this case is blind trie (a cache-oblivious trie data structure used to handle string dictionaries) and the strings are then sorted using the cache-oblivious list ranking [4] technique.

Tried linear hashing is a combination of tried hashing and linear hashing. It combines the best of linear hashing and tried hashing. Linear hashing allows the data file to expand gracefully and on the other hand, trie hashing uses a trie structure for efficient handling. Tried linear hashing combine these two concepts and it reduced the number of disk access by 5% for a load factor of 0.6 and 62% for a load factor of 0.9 [3]. Tried linear hashing is better than other hashing because it maintains the order of the

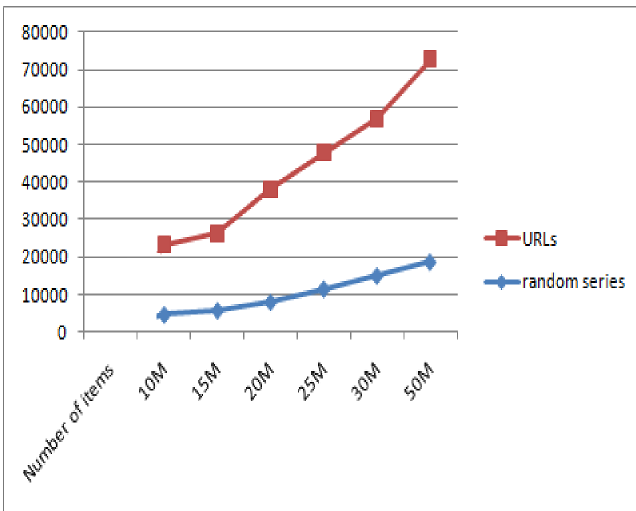
keys to be hashed which in this case can be the length of the longest common prefix. In our algorithm we experimented with various size of buckets. The hashing does not sort the strings but they help in mapping the strings with the common prefix together.

The algorithm presented in [7] proceeded in two parts: first includes signature reduction done using hashing and in second the strings are sorted using the list ranking technique. The signature reduction is done to make a compact trie. After signature reduction, an ordered trie is constructed. But in our algorithm during the hashing we create an ordered compact trie i.e., a Blind trie and after creating that list ranking algorithm is performed on it.

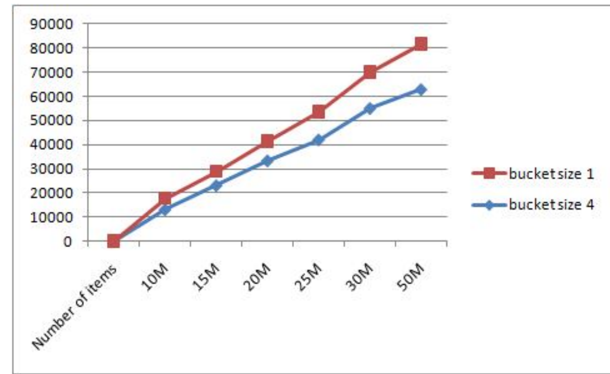
We performed experiment using different size of buckets for tried linear hashing and got the following results.



(a) Result for bucket of size 1(measured the running time in milliseconds)



(b) Result for bucket of size 4 (measured the running time in milliseconds)



(c) Comparison of different bucket sizes for random strings

Fig. 7: Showing the experimental results.

5. Conclusion

This paper briefs about various efficient algorithms and data structures which are cache-oblivious and also presents a new and improved string sorting algorithm which uses the concept of tried linear hashing, blind tries and list ranking technique for sorting strings. The cache-oblivious algorithms are proving to be more beneficial than cache-aware algorithms in some cases and are being improved with time. These if implemented properly can help in improving the efficiency of algorithms.

6. References

1. Aggarwal, A., & Vitter, J. S. (1988). The input/output complexity of sorting and related problems. Communications of the ACM 31,9 , 1116-1127.
2. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company .
3. Ang, C. H., Tan, S. T., & Tan, T. C. (1998). Tried Linear Hashing. ASIAN .
4. Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., & Munro, J. I. (2002). Cache-oblivious priority queue and graph algorithm applications. In proceedings of the Symposium on Theory of Computing, ACM, New York , 268-276.
5. Bender, M. A., Demaine, E. D., & Farach-Colton, M. (2000). Cache-oblivious B-Trees. In Proc. 41st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 339-409.
6. Brodal, G. S., Fagerberg, R., & Jacob, R. (2002). Cache-oblivious search trees via binary trees of small height. In Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms , 39-48.

7. Christiansen, T. b. (n.d.). Algorithms for string sorting in external emeory. Master's Thesis .
8. Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-Oblivious Algorithms (extended abstract). Proceedings of the 40th Annual Symposium on Fondations of Computer Science, IEEE Computer Society Press, 285-297.
9. Hennessy, J. L. (2003). Computer Architecture: A Quantitative Approach. 3th Edition, Morgan kaufmann Publishers Inc.
10. Knuth, D. E. (1973). Sorting and Searching. volume 3 of The Art of Computer Programming. Addison-Wesley, Reading, MA .
11. Moore, G. (1965). Cramming more components into integerated circuits. Electronics magazine 38 , 114-117.
12. Pagh, R., Wei, Z., Yi, K., & Zhang, Q. (2010). Cache-oblivious hashing. In Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database system of data .
13. Prokop, H. (June 1999). Cache-oblivious Algorithms. Master's Thesis, Massachusetts Institute of Technology, .
14. Yotov, K., Roeder, T., Pingali, K., Gunnels, J., & Gustavson, F. (2007). An experiment comparison of cache-oblivious and cache concious programs. In proceedings of the Symposium on parallel Algorithms and Architectures, ACM, New York , 93-104.
15. Kamal, N, & JiaHui, Z. Cache-oblivious Algorithm, A Parallel Project Report. Singapore MIT Alliance.