IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Optimization of Cache Memory Using Unrolled Linked List

Neha Jain*1, Kamlesh Lakhwani [2]
[*1,2] M. Tech., Suresh Gyan Vihar University, Jaipur, Rajasthan
kinshika@gmail.com

Abstract

Data structures play an important role to reduce the cost of memory access. First array was introduced which gives fast cache-access, as elements are stored in consecutive places. But it has disadvantage also, like, wastage of memory, don't allow insertion at run-time, and most important takes large number of shifting operations to insert or delete an item at any other position excluding last position.

After that Linked-List was introduced to overcome disadvantages of array. It has advantages of quick insertions and incremental growth. At run-time new data can be inserted and avoid wastage of memory. But it also has some disadvantages, like requirement of extra memory space to store references. For N elements N references are assigned, one for each element.

Special type of Linked-List is introduced which combines the cache advantages of an array but the quick insertions and incremental growth of a linked list. This is Unrolled Linked-List. Actually data field stores an array in place of a single element.

As Unrolled Linked-list is a cache-sensitive data structure, which increases the speed of memory access by using arrays. We'll implement its all basic operation along with its applications. Basic operations are insertion, deletion, searching, traversing and sorting. Like array and Linked-

List it also has applications to implement many other elementary data structures like stack and queue.

In future may be more cache-sensitive and cache-obvious data structure can be defined and implemented. Also NP-Hard and NP-Complete problems can be solved using these data structures like unrolled linked-list.

Keyword: - Cache Memory, Linked List, Unrolled Linked List, Memory Access

Introduction

Today a memory access can be hundreds of times more costly than an arithmetic operation. Machine designers have tried to reduce this cost by using special hardware and software tools, multiple levels of cache, non-blocking caches, dynamic instruction scheduling, etc. These techniques have only been partially successful for pointer-manipulating programs. Memory caches are the ubiquitous hardware solution to this problem. A memory cache is a small and fast memory that store recently accessed data items and attempt to intercept and satisfy data requests without accessing main memory. In the beginning, a single level of cache is sufficed, but the increasing performance gap requires two levels of caches today and three in the near future. In addition to caches, a variety of hardware and software techniques have been developed and implemented to reduce the cost of memory access. Except these techniques, many program's execution time is dominated by the latency of memory references. Random-Access Memory Model (RAM) is generally used by programmers to understand and design a data structure and algorithm.

From a software perspective, programming languages also plays an important role. Early languages like FORTRAN and ALGOL, used mainly for scientific applications, did not support pointers. Applications written in these languages stores their data in array. Subsequent languages such as Simula, Pascal, C and C++ supported pointers. Many applications written in these languages, such as databases and operating systems, make extensive use of pointer structure to store data.

In addition to all these techniques, data structures are also important to speed up memory access. As we already know that Arrays are linear data structure and very efficient to perform insertion and deletion operation at last position. One advantage is also that they are very easy to implement. To insert an element at middle or starting requires O(N) time, which is very costly.

Then another type of data structures is Linked-List. This data structure is specially pointers. A pointer is a variable refer to the other variable. Pointers are used to allocate memory at runtime. They are frequently used in applications in which the data requirements are not known until the program executes or varies widely during execution. In insertion at middle there is no backward shifting just finding middle position we can insert the new node. Similarly with deletion, there is no forward shifting. Linked-lists are

important to reduce the cost of memory access for particular applications. We all know the implementation of these data structures from a simple Array to the circular linked-list in any programming language like C, C++, and java etc.

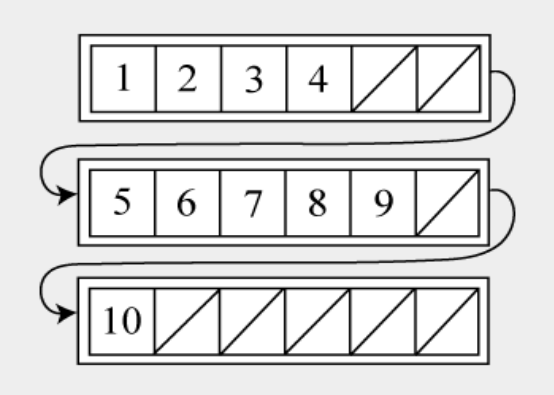
Objective

1. Implementation of elementary data structures.
2. Overcomes disadvantages of Linked-List, by storing multiple elements in each node.
3. It combines cache advantages of an array with quick insertion and incremental growth of Linked-List.

Unrolled Linked-List

Modern PCs have multi-level cache hierarchies that make traversing an array (visiting the elements in order) very fast. Cache hits are so fast that in cache-sensitive analysis they are considered "free"; we only count cache misses. If a cache line has size B, then the number of cache misses is about n/B. A linked list, on the other hand, requires a cache miss for every node access in the worst case. Even in the best case, when the nodes are allocated consecutively in order, because linked list nodes are larger, it can require several times more cache misses to traverse the list. Traversing of an array is faster than a linked-list, having both 60 million integers. Don't ignore linked-list just yet, though. We need something with the cache advantages of an array but the quick insertions and incremental growth of a linked list.

Here comes the role of "Unrolled Linked-List". In simple terms, an unrolled linked list is a linked list of small arrays, all the same size N. Each is small enough that inserting or deleting from it is quick, but large enough so that it fills a cache line. An iterator pointing into the list consists of both a pointer to a node and an index into that node.



Methodology

Basic Operations

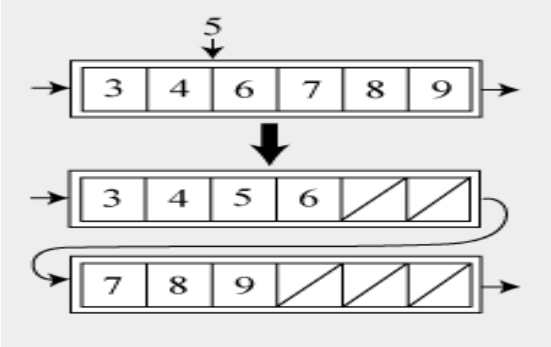
1. Insertion
2. Deletion
3. Traversing
4. Searching

Insertion

If there is space left in the array of the node in which we wish to insert the value, we simply insert it, requiring only O(N) time. If the array already contains N values, we create a new node, insert it after the current one, and move half the elements to that node, creating room for the new value. Again, total time is O(N).

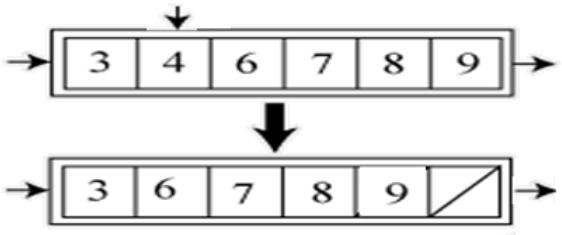
Insertion in Unrolled Linked-List

Deletion



Deletion is similar; we simply remove the value from the array. If the number of elements in the array drops below N/2, we take elements from a neighboring array to fill it back up. If the neighboring array also has N/2 values, then we merge it with the neighboring array instead. Those familiar with B-Trees may note some similarities in these operations.

One small issue with unrolled linked lists is keeping track of the number of elements in each array. One easy and space-efficient way to deal with this, where applicable, is to use some reserved "null" value to fill unused array slots. By aligning the nodes, sometimes the number of elements can be stored in the low bits of the pointers. Otherwise it adds a bit of overhead per node.



Deletion operation in Unrolled Linked-List

Work Plan

Implementation of Insertion operation in C:

I am familiar with C language, that's why in my dissertation I am using C. Yet I have only written a C code for insertion operation in Unrolled Linked-List. In this insertion case I have assumed that, when a node in which array is full, then a new node will be added after that node and new elements will be inserted in that new node.

Conclusion

Data structures play an important role to reduce the cost of memory access. First array was introduced which gives fast cache-access, as elements are stored in consecutive places. But it has disadvantage also, like, wastage of memory, don't allow insertion at run-time, and most important takes large number of shifting operations to insert or delete an item at any other position excluding last position.

After that Linked-List was introduced to overcome disadvantages of array. It has advantages of quick insertions and incremental growth. At run-time new data can be inserted and avoid wastage of memory. But it also has some disadvantages, like requirement of extra memory space to store references. For N elements N references are assigned, one for each element.

Special type of Linked-List is introduced which combines the cache advantages of an array but the quick insertions and incremental growth of a linked list. This is Unrolled Linked-List. Actually data field stores an array in place of a single element.

I have implemented its insertion operation. In this each node contains an array of size N. To insert an element there are two conditions, i) if in node array have space then that element will be inserted easily, ii) if array is full then a new node will be created and inserted after the current node and element will be inserted in that node.

One small issue with unrolled linked lists is keeping track of the number of elements in each array. One easy and space-efficient way to deal with this, where applicable, is to use some reserved "null" value to fill unused array slots. By aligning the nodes, sometimes the number of elements can be stored in the low bits of the pointers. Otherwise it adds a bit of overhead per node.

Future Scope

As Unrolled Linked-list is a cache-sensitive data structure, which increases the speed of memory access by using arrays. In future I'll implement its all basic operation along with its applications. Basic operations are insertion, deletion, searching, traversing and sorting. Like array and Linked-List it also has applications to implement many other elementary data structures like stack and queue.

Insertion operation can be modified, If there is space left in the array of the node in which we wish to insert the value, we simply insert it, requiring only O(N) time. If the array already contains N values, we create a new node, insert it after the current one, and move half the elements to that node, creating room for the new value. Again, total time is O(N). Deletion is similar to insertion; we simply remove the value from the array. If the number of elements in the array drops below N/2, we take elements from a neighboring array to fill it back up. If the neighboring array also has N/2 values, then we merge it with the neighboring array instead.

In future may be more cache-sensitive and cache-obvious data structure can be defined and implemented. Also NP-Hard and NP-Complete problems can be solved using these data structures like unrolled linked-list.

References

[1] M. Fomitchev. Unrolled linked lists and skip lists.Master's thesis, York University, October 2003.

[2] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In DISC 2001, pages 300–314.

[3] M. R. BROWN AND R. E. TARJAN, Design and analysis of a data structure for representing sorted lists, SIAM J. Comput. 9 (2000), 594-614.

[4] Shimomura, T. & Isoda, S. 1991, "Linked-list visualization for debugging", IEEE Software 8(3), 44-51.

[5]http://en.literateprograms.org/Unrolled_linked_list_(C_Plus_Plus)

[6] http://blogs.msdn.com/b/devdev/archive/2005/08/22/454887.aspx

[7]http://locklessinc.com/articles/optimization/