Effective Reducing Memory Latency Using Data Prefetch Mechanisms

Akhil Khare, Ahmed Khodair

Abstract— During linked data structures (LDS) traversals, prefetching improves the performance by reducing memory latency. We will discuss about the jump pointer prefetching which hides additional load latency by using an extra pointer to Prefetch objects further than a single link away. Jump pointers can be implemented in Binary tress by adding jump pointers at creation time and in LDS by adding jump pointers at traversal time. Prefetch Arrays are also used to store jump pointers. It has two approaches hardware and software. Both the approaches have highly improved the performance of prefetching with the use of jump pointers. Prefetching in pointer-based codes (java programs) is difficult because separate dynamically allocated objects are disjoint, and the access patterns are thus less regular and predictable. However, according to experimental results, the largest performance improvement is 48% with jump-pointers in java programs, but consistent improvements are difficult to obtain.

I. INTRODUCTION

The linked data structures or the LDS traversal often takes place in loops or recursion. There occurs a problem with LDS traversal called as Pointer-chasing problem which takes place if the data is not found in the cache. This problem occurs because LDS consists of the chains of loads which are data dependent on each other and form the links of the LDS as a result of which parallel data prefetching becomes limited and load latency increases. The LDS load latency can be hidden and the performance of LDS traversal can be improved with the help of Prefetching. Address prediction based techniques can calculate the address and prefetch the desired arbitrary node but it has its drawback of not predicting the correct address on a regular basis. The scheduling technique prefetches nodes serially but hides the induction (l->l->next) load latency by scheduling it early in the iteration. However, it is not effective if the work between the iterations is not enough to overcome the latency. The inclusion of jump pointers in this technique can highly improve the performance of the LDS traversal. The following figure explains how jump pointers can be used to leverage the work of multiple iterations. The introduction of Jump pointers have made it possible to overcome the shortcomings of the techniques which required either to predict the traversal path a priori or the work between two consecutive LDA accesses to be enough to overlap the latency. Before discussing further about jump pointers we must discuss about greedy prefetching too. Greedy prefetching prefetches directly connected object(s) during each iteration of a loop or recursive function call.

Jump pointers are those which point to the nodes which are located more than a single link away. Jump-pointer prefetching may hide additional latency by using an extra pointer to prefetch objects further than a single link away. Jump-pointers are a flexible mechanism for linked data structures because we can prefetch arbitrary objects and not just directly connected objects. Jump-pointer prefetching is potentially able to tolerate any amount of latency by varying the distance between the two objects. Jump-pointer prefetching may also reduce the number of prefetches, yet still remain effective. Furthermore, jump-pointer prefetching does not prefetch null objects at the leaf nodes in a binary tree.

Jump pointer prefetching can be implemented either by a compiler which automates jump-pointer prefetching by inserting code to create and update the jump-pointers as well as inserting prefetch instructions at appropriate places in the program or with the use of jump pointers and prefetch arrays. Jump pointers point to nodes which are not adjacent in a linked list. Prefetch arrays consist of a number of jump

---

**Figure 1.** Hiding LDS load latency. (a) Exposed induction load latency can be hidden by (b) scheduling it early in an iteration. (c) This approach is ineffective if a single iteration has insufficient work. (d) Jump-pointers can leverage the work of multiple iterations.

**Figure 2:** showing greedy prefetching
pointers located in consecutive memory. These are used to aggressively prefetch several nodes in parallel that potentially will be visited in successive iterations.

II. PREFETCHING

Prefetching in general means bringing data or instructions from memory into the cache before they are needed. When an application needs data that was prefetched, instead of waiting for the data from memory, it can grab it from cache and keep right on executing. There are actually two main ways in which prefetching can occur: initiated by hardware or initiated by software.

A. Hardware Prefetching

Hardware prefetching is implemented by your processor and will be different depending on which processor you use. Most recent Intel processors have several different hardware prefetchers. The Core™ i7 processor and Xeon® 5500 series processors, for example, have some prefetchers that bring data into the L1 cache and some that bring data into the L2. There are also different algorithms – some monitor data access patterns for a particular cache and then try to predict what addresses will be needed in the future. Others use simpler algorithms, such as fetching 2 adjacent cache lines.

B. Software Prefetching

Software prefetching is implemented by software developers. It involves identifying when your application will need a particular set of data, then using special prefetch instructions to tell the processor to get this data in advance.

III. LINKED DATA STRUCTURES

Linked data structures (LDS) are those data structures which are connected to each other with the help of pointers to the either the next node or other nodes. They are common in many applications, and their importance is growing with the spread of object-oriented programming. There are two kinds of prefetching algorithm implemented in LDS known as Greedy Prefetching and Jump Pointer Prefetching. Regular traversals of a linked data structure are identified by a recurrent update to a pointer variable. A recurrent update is a field assignment of the form o = o.next that appears within a loop (intraprocedural) or recursive call (interprocedural), as shown in the examples below. Each execution of the assignment updates the pointer variable with a new object of the same type, either directly or by using a temporary.

DIRECT
while (o != null)
{
  o.compute();
o = o.next;
}

USING A TEMPORARY
while (o != null)
{
  o.compute();
t = o.next;
...;
o = t;
}

Fig 3: Linked Data Structures (Doubly Linked List)

A. Greedy Prefetching

In this section, greedy prefetching algorithm is described which prefetches directly connected objects in recurrent accesses. Figure below shows simple class definitions for a singly linked list, a doubly linked list, and a binary tree. Each class contains a sum method which adds the elements in the data structure. In the list examples, we insert a prefetch instruction for the next object in the linked list. We cannot prefetch objects further ahead because we do not know the address of future objects.

Fig 4: Greedy prefetching in a doubly linked list

B. Jump Pointer Prefetching

Jump-pointers are a flexible mechanism for linked data structure because we can prefetch arbitrary objects and not just directly connected objects. Jump pointer prefetching is potentially able to tolerate any amount of latency by varying the distance between the two objects. Greedy prefetching restricts the amount of latency tolerance by prefetching direct links only, but does not require an additional field to store the jump-pointer. Jump pointer prefetching may also reduce the number of prefetches, yet still remain effective.

The compiler automates jump-pointer prefetching by inserting code to create and update the jump-pointers as well as inserting prefetch instructions at appropriate places in the program.

Fig 5: Jump-pointer prefetching in a doubly linked list
Jump pointer prefetching can be implemented either by a compiler which automates jump-pointer prefetching by inserting code to create and update the jump-pointers as well as inserting prefetch instructions at appropriate places in the program or with the use of jump pointers and prefetch arrays. Jump pointers point to nodes which are not adjacent in a linked list. Prefetch arrays consist of a number of jump pointers located in consecutive memory. These are used to aggressively prefetch several nodes in parallel that potentially will be visited in successive iterations.

IV. CREATING JUMP POINTERS

As discussed above, the jump pointers are used to reduce the memory latency during the process of prefetching. In LDS, this is made possible by overlapping the latency involved in node access by the work between the two iterations. For this purpose, the distance (in dynamic nodes traversed) between the home and the target nodes should be proportional to the target node access latency. Take for example, if each node consists of 5 cycles of work and the node access requires 25 cycles, then the home node of the jump pointer should be 5 nodes ahead of the target node. The distance between the two nodes has to be accurate because

1.) If the distance is too short, only part of the target access latency will be hidden.
2.) If the distance is too long, the prefetch block will be evicted before it can be used.

Since gathering the ideal distance between nodes is a daunting task, a prefetch distance (prefD) is selected which is usually the maximum or the average required distance per node. The jump pointers are then set at a distance of prefD before the target nodes. This is easily accomplished using a queue of length prefD. On LDS creation, or first traversal, a queue maintains the last prefD node addresses. As each new node is added (traversed) a jump pointer is created with the code described below. The code shown previously is for building jump-pointers in a binary tree object at creation time. The circular queue, jumpQ, maintains a list of the last n objects allocated. When a new object allocation occurs, a jump-pointer is created from the object at the head of jumpQ to the new object. Then, the new object is inserted at the end of jumpQ, and the circular queue index is advanced [3, 8].

The efficiency of prefetching is determined by the following four factors:-
1.) Time to perform the whole loop body i.e., work.
2.) Branching factor of the LDS BranchF. In case of a binary tree it is two.
3.) The number of nodes traversed i.e., chainL or chain length.
4.) The latency of the load or the prefetch i.e., Latency [1].

$LHC_{pf} = \begin{cases} 
1 & \text{if } Work \geq Latency \\
\frac{Work}{Latency} & \text{if } Work < Latency
\end{cases} \quad (1)$

For hiding latency while prefetching nodes when Work<Latency, the distance of the next node needs to be calculated. This distance is referred as prefD. To attain this
goal, jump pointers are required. These jump pointers are the extra pointers in the nodes which point to the nodes that are prefD iterations ahead. The drawback of this technique is that the nodes that are prefD-1 distance ahead cannot be perfected. Hence, if chainL<prefD, then the prefetch hiding capability will be zero. If instead chainL>=prefD, ignoring the effect of LHC on the first prefD-1 load misses and branchF=1 there is only one traversal path, thus LHC=1 if prefD is set properly. But if the branchF is more than 1, then it is assumed that every tree node is prefetched with the same probability and that the tree is traversed depth first until the first leaf node is reached. The probability that the tree node will be prefetched is

\[
\frac{1}{\text{Branch}\text{F}^\text{PrefD}} \text{, thus } LHC = \frac{1}{\text{Branch}\text{F}^\text{PrefD}}. \text{ The effectiveness is thus as follows:}
\]

\[
LHC_{\text{app}} \leq \begin{cases} 
\frac{1}{\text{Branch}\text{F}^\text{PrefD}} & ; \text{ChainL} \geq \text{PrefD} \\
0 & ; \text{ChainL} < \text{PrefD}
\end{cases}
\]

(2)

If the traversal path is known beforehand then the LH can be made equal to 1 as the jump pointers can be initialized to point down the correct path.

VI. CONCLUSION

Software prefetching is an efficient technique to tolerate long memory latencies. Software prefetching has to be accurate and timely in order to be effective. This approach may use compile-time information to perform sophisticated prefetching, whereas the hardware scheme has the advantage of manipulating dynamic information. The hardware automatically creates and updates jump-pointers and generates addresses for and issues prefetches. The overhead due to the extra prefetch instructions and associated computations is substantial in the software approach and can offset the performance gain of prefetching. Our experimental results show that the new solution is very attractive in reducing the data access penalty without incurring much overhead.

REFERENCES


AUTHOR’S PROFILE

Mr. Akhil Khare is working as an Assistant Professor in Information Technology Department at Bharati Vidyapeeth Deemed University College of Engineering, Dhanawadi, Pune India. He was awarded his Master of Technology Degree from Government Engineering College Bhopal in 2006. His areas of interest are Computer Network, Software Engineering and Multimedia System. He has Eighth years experience in teaching and research. He has published more than twenty research papers in journals and conferences. He has also guided ten postgraduate students.

Ahmed Khodair from Information Technology Department at Bharati Vidyapeeth Deemed University College of Engineering, Dhanawadi, Pune India. His areas of interest are Software Engineering and Multimedia System. He has one years experience in teaching and research.