Oriental Journal of Computer Science & Technology Vol. **3**(1), 45-52 (2010)

Software prefetching using jump pointers in linked data structures

ARUSHI ARORA, SWATI PRIYA and AKHIL KHARE

Department of Information Technology,
Bharati Vidyapeeth College of Engineering, Pune - 411 043, (India).

(Received: April 12, 2010; Accepted: June 04, 2010)

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.

Keywords: Jump pointers, Linked Data Structures, Software prefetching.

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. A*ddress 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 (I=I->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.

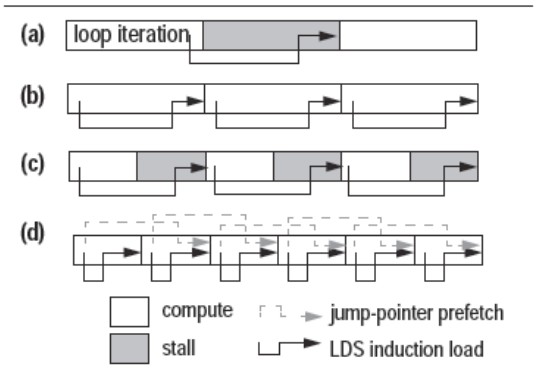


Figure 1. Hiding LDS load latency. *(a) Exposed induction load latency can be hidden by (b) scheduling it early in an iteration. (c) This approach is ineffective if a single iteration has insufficient work. (d) Jump-pointers can leverage the work of multiple iterations.*

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.


```
int sum (Dlist l) {
  int s = 0;
  while (l != null) {
    prefetch(l.next)
    s =+ l.data;
    l = l.next;
  }
  return s;
}
```

Fig showing greedy prefetching.

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.

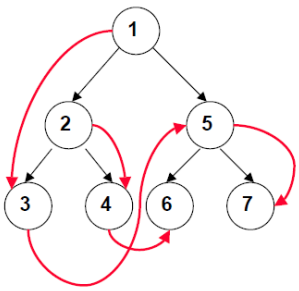
Jump pointer prefetching can be implemented either by a compiler which automates jump-pointer prefetching by inserting code to create and update the jump-pointers as well as inserting prefetch instructions at appropriate places in the program or with the use of *jump pointers* and *prefetch arrays*.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

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 s 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 jumppointer is created with the node at the head of the queue as its home and the current node as its target. The current node is then enqueued at the tail of the queue, while the home node at the head is removed. Jump pointers for binary trees can be created with the code described below.



Creation Order	1234567 OR 7654321
Traversal Order	1234567

Jump Pointer Prefetching-binary Tree Traversal

Fig 2.

```
Class Tree
{
Int value;
Tree left;
Tree right;
Tree prefetch;
}

Tree createTree(int l)
{
If(l==0) return null;
 Else
{
Tree n=new Tree();
jumpObj=jumpQ[i];
jumpObj.prefetch=n;
jumpQ[i++%size]=n;
Tree left=createTree(l-1);
Tree  right=createTree(l-1);
n.left=left;
n.right=right;
return n;
}
}
```

The code shown above 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 ics 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].

Adding Jump Pointers at Creation Time

Jump pointers when added at object creation time in data structures with regular access patterns, minimizes run time cost because the jump pointers are created only once. But as in Fig 2 the limitation that the creation must be preorder, beginning with either the left or the right sub tree makes it difficult to create effective jump pointers at creation site. If the jump pointers are built bottom up, then they will not be useful. Also if a program frequently updates a linked structure containing jump pointers then the original jump pointers become invalid.

Adding Jump Pointers During Traversal

For programs that frequently traverses and updates the linked structure, building jump pointers is very effective. Also here, the code to create jump pointers appears locally. However, this approach becomes less effective when the traversal pattern of the LDS changes e.g., accessing a list in one direction followed by an access in the reverse direction, due to the overhead of maintaining the jump pointer queue[3].

Performance Measurement

The performance measurement is being carried out for LDS which is a list or a tree which is traversed depth first.In a particular LDS traversal algorithm, there is present a loop or a recursion and a node in the LDS is fetched and some work is performed using the data found in the node. In the Fig 2.1,the tree is traversed depth first. In each iteration, a node is visited ,some work is performed and the next node is fetched. The loop is repeated until there are no more nodes. An important observation is that a load that fetches the next node is dependent on each of the loads that fetched the former nodes. Hereafter, these loads are being referred to as the *pointer chase loads*.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(Latency Hiding Capability) is the fraction of the pointer-chase load latency that is hidden by prefetching. For the prefetch to be fully effective, Work <= Latency must hold. In this case, a prefetch would be issued at the beginning of an iteration and it would be completed when the iteration ended and LHC = 1. However,if Work < Latency only a part of the latency would be hidden as shown in the equations below.

$$LHC_{gr} = \begin{cases} 1 & ; Work \geq Latency \\ Work/Latency & ; Work < Latency \end{cases} \quad (1)$$

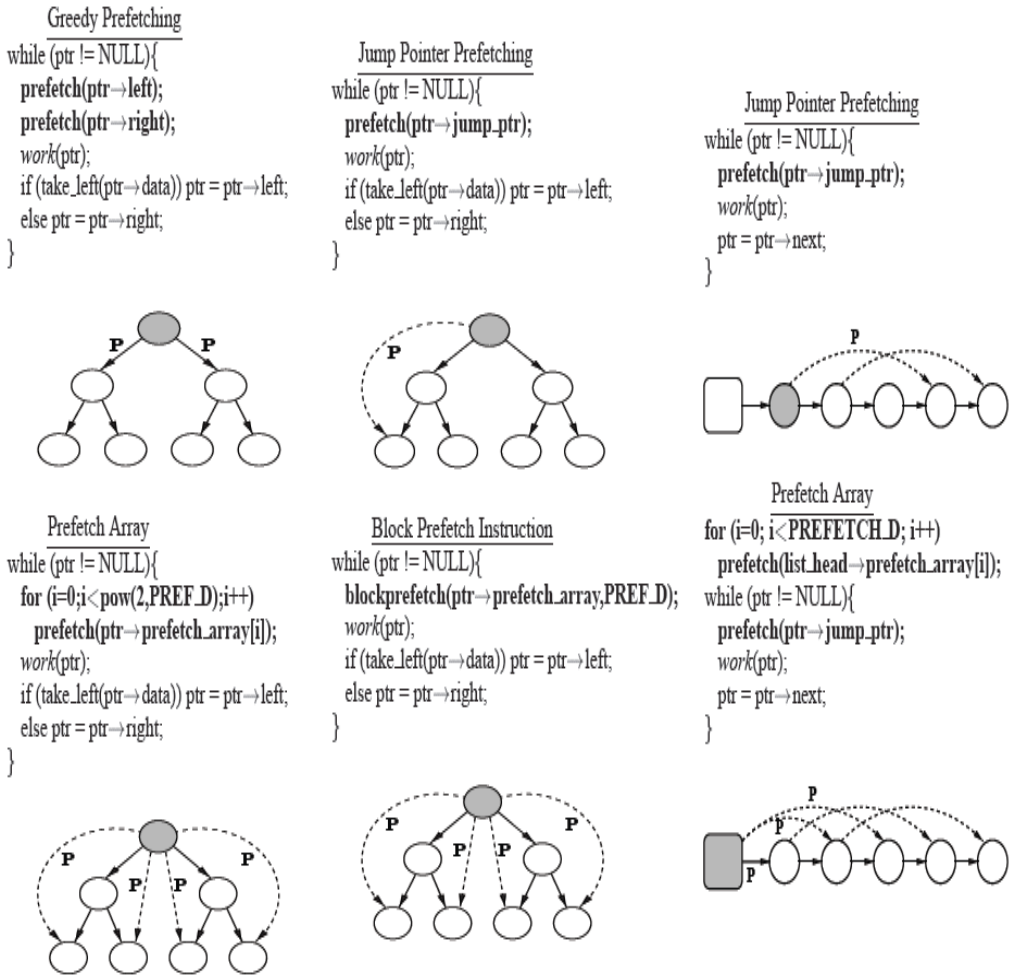


Fig 2. The prefetch launches by the jump pointers in a binary tree and a linked list.P denotes the jump pointer launches and PREFETCH_D is the fixed distance *fd* discussed.

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 prefetched. 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 $1/BranchF^{PrefD}$, thus $LHC = 1/BranchF^{PrefD}$. The effectiveness is thus as follows:

$$LHC_{jpp} \leq \begin{cases} \frac{1}{BranchF^{PrefD}} & ; ChainL \geq PrefD \\ 0 & ; ChainL < PrefD \end{cases} \quad (2)$$

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

Prefetch Arrays

The jump pointers are stored in consecutive memory in an array called as the *prefetch array*.This array is located in the node so that when a node is fetched into the cache the corresponding prefetch array is likely to be located on the same cache line, thus most of the time there will be no extra cache misses when accessing the prefetch array. The prefetch array is then used in every iteration to launch prefetches for all the nodes a number of iterations away. An example code how this is implemented for a binary-tree is shown in Figure 3.

When BranchF = 1, jump pointer prefetching has the disadvantage that it does not prefetch the first *fd*-1 nodes as shown in Figure 3. This is very ineffective for short LDS.A technique has been devised where the prefetch array is created at the head of the LDS which points to the first fd-1 nodes that the regular jump pointers do not point to. This prefetch array is used to launch prefetches before the loop is entered that traverses the list. How this is implemented can be seen in Figure 2. Once the loop is entered, prefetches are launched as in jump pointer prefetching. Note that for all other types of LDS, the main prefetching principle is the same as the binary tree.

LHC In Binary Trees

The LHC of our method will be different if we are traversing lists or traversing a tree, because there are no jump pointers pointing to the first nodes in a tree as there are in a list. The rationale behind this is that short lists are much more common (in hash tables for example) than short trees and that the top nodes of a tree will with a high probability be located in the cache if the tree is traversed more than once. On the other hand, if this is not true, a prefetch array could be included that points to the first few nodes in the tree, in the same manner as for lists. For a tree, as we prefetch all possible nodes at a prefetch distance prefD the effectiveness will be 1 when chainL>=*prefD* (ignoring the misses of the first *prefD*-1 nodes). However,if chainL<*prefD* there will be no prefetches.

$$LHC_{pa\_tree} = \begin{cases} 1 & ; ChainL \geq PrefD \\ 0 & ; ChainL < PrefD \end{cases} \quad (3)$$

For a list, the effectiveness will also be 1 when chainL>=prefD if we ignore the misses to the first prefD-1 nodes. However, we also aggressively prefetch the first nodes in an LDS. This means that we get some efficiency even when chainL<*prefD*. The LHC of the prefetch fetching the first node will be Min(1,work/latency).The prefetches for the following nodes will by that time have arrived, and the LHC of those will be one. The equations are summarised below.

$$LHC_{pa\_list} = \begin{cases} 1 & ; ChainL \geq PrefD \\ \frac{\min(1, \frac{Work}{Latency}) + PrefD - 1}{ChainL} & ; ChainL < PrefD \end{cases} \quad (4)$$

In using our prefetching scheme, we have three limiting factors: memory overhead, bandwidth overhead, and instruction overhead both due to prefetching and rearranging of jump pointers and prefetch arrays when inserting/deleting nodes from the LDS. The memory overhead consists of space for the prefetch arrays and the jump pointers if used. If there are branchF possible paths for each node and the prefetch distance is prefD then the number of words each prefetch array occupies is BranchF_prefD.If *prefD* is large and BranchF >1. the number of nodes that need to be prefetched soon gets too numerous, and both the memory, bandwidth and instruction overhead will be too high for *Prefetch arrays* to be effective.

Prefetch Arrays: A Hardware Approach

Hardware-based prefetching techniques do not cause the instruction overhead associated with the use of explicit fetch instructions. Hardware-based prefetching, with the help of cache, can dynamically handle prefetches at run-time without compiler intervention. Software-directed approaches rely on compiler technology. However, without the benefit of compile-time information, hardware prefetching relies on guessing about future memory-access patterns based on previous patterns. Thus the incorrect guessing will cause the memory system to bring unnecessary blocks into the cache. These unnecessary prefetches do not affect correct program behavior, but instead cause cache pollution and consume memory bandwidth. Hardware Jump Pointer Prefetching (JPP) faces various challenges in finding jump-pointer storage. For a hardware-only implementation, the

Dependence Based Prefetch (DBP) mechanism is used with structures that direct jump-pointer creation (storage) and prefetching (retrieval). Dependence-based prefetching dynamically identifies loads that access linked data structures. This mechanism implements *chain jumping*: restricting jump pointer prefetching to recurrent "backbone" loads and using DBP to automatically chain prefetch "rib" loads. This solution automatically provides queue jumping wherever necessary. This mechanism is simple to implement in hardware and handle most programs. Full and root jumping are not implemented, due to difficulties with finding jump-pointer storage and also we cannot rely on them in case of understanding high level programs.

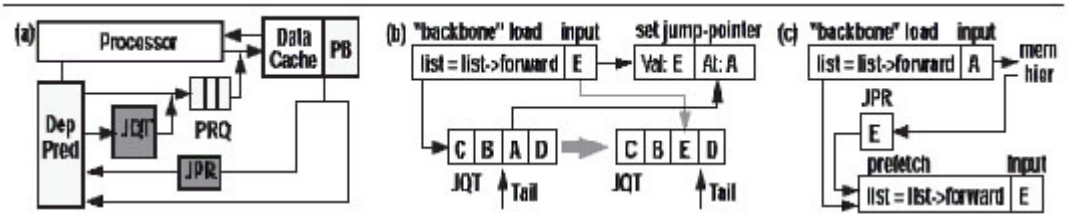


Figure 3. Hardware JPP. (a) Block diagram with DBP specific parts in light gray and JPP components in dark gray. (b) Installing jump-pointers: the Jump Queue Table (JQT) entry contains the previous four input addresses of the load **list = list->forward**. *When a new instance commits, it creates a jump-pointer from the node at the queue tail, A, to the current node, E. It then updates the JQT, advancing the queue. (c) Jump-pointer prefetching: As a "backbone" load issues, the jump-pointer in the corresponding home node is placed in the JPR and used to launch a prefetch.*

For jump-pointer creation, queue method in hardware is implemented. Each *static* load is identified as being recurrent ("backbone") and is associated with a queue that keeps the note of its most recent input addresses. Address queues for the set of active recurrent loads are stored in the Jump Queue Table (JQT). As soon as an instance of a recurrent load commits, it accesses the JQT and creates a jump-pointer from the node sitting at the head of the queue to the node corresponding to its own input address. This process is illustrated in Figure 3(b). **list = list->forward** creates a jump-pointer from the node visited four hops ago, **A**, to the current node, **E**. A request is generated for storing this jump-pointer while the queue is updated indicating the access of the current node. Jump-pointer retrieval and prefetch initiation is a more delicate process which is explained Figure 3(c). Whenever an LDS "backbone" load issues, the jump-pointer residing at the corresponding home node is placed into a special non-architected location called the Jump pointer Register (JPR). A jump-pointer prefetch is created using a speculative instance of the load with the JPR *value* as its input. The main issue in implementing hardware jump-pointers is not which pointers to create, but rather where to store them.

EXPERIMENTAL RESULTS

Various kernels have been evaluated according to their efficiency when prefetching is implied. The effect of prefetching on the execution time of tree and list traversals is being discussed.

Kernel	Greedy Prefetch	Jump Pointers	Prefetch Arrays	
			Software	Hardware
mst	100%	42%	75%	75%
mst.long	100%	82%	82%	82%
health	100%	97%	97%	97%
DB.tree	57%	24%	32%	32%
treeadd	77%	97%	62%	62%
perimeter	78%	99%	96%	96%

The kernel *Mst* is a hash table benchmark. Here the chain L is between two and four and the Work<<Latency. Here since the work is low, greedy prefetching improves the performance by only 2% and jump pointer only by 1% because chain L< fd most of the time. In PA, part of the latency of the first nodes in the list will be hidden and thus, performance is seen in the short lists as well. The software approach improves performance by 20%. While the hardware approach improves it by 22%. The second kernel *Mst. long* has a larger chain L but otherwise it is the same as mst. This leads to the conclusion that jump pointer prefetching is more

effective here.It improves the performance by 35%.However, PA improves the performance by 47% and 48% respectively. The kernel *Health* has the following properties: these are not small programs and LDS nodes are inserted and deleted frequently during the execution. The second fact should indicate that jump pointer prefetching and prefetch arrays should be less effective as they incur overheads for insert and delete operations. However, it is seen that jump pointer prefetching improves the performance by 27% and PA with 38% and 39%.The kernel DB. tree is taken from a database server, and is a depth-first traversal of a binary index tree in search of a leaf node. The important aspect of this kernel is that the traversal path is not known a priori, thus the jump pointers in jump pointer prefetching is set to point down the last path traversed, thus the high instruction overhead for this technique compared to the other. But jump pointer is not a good option for programs where traversal is not known a priori. It even increases the memory stall time due to cache misses on jump pointer references that most of the time are useless.Also, the software PA suffers from a high instruction overhead from the issuing of the prefetches and only manages to improve the performance by 3%. The best prefetch approach for this kernel is hardware PA which improves the performance by 28%. Greedy prefetching improves the performance by 15% as the value of Work is close to the value of Latency. In the kernel *Treeadd* ,the traversal path is known a priori. Thus, jump pointer provides a performance increase here. Also the memory stall time can be removed almost entirely by changing the prefetch distance to a higher value, thus jump pointer prefetching can outperform all other techniques for *treeadd*. However, jump pointer prefetching needs to adjust the prefetch distance for vary ing memory latencies, while prefetch arrays gives a 40%execution time reduction, for this kernel, without any tuning. The last kernel is *perimeter* that traverses a quadtree in which the traversal path is known a priori. Both prefetch array approaches cannot improve the performance as there are far too many prefetches that need to be launched when each new node is entered. The best approach to use should be jump pointer prefetching after an adjustment of the prefetch distance to a higher value, which would reduce the memory stall time even more[1].

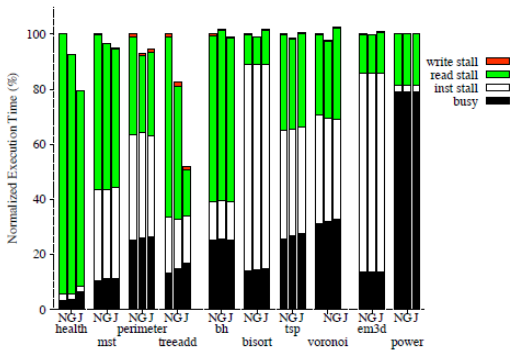


Fig 5 Performance measurement in ten programs to derive the maximum efficiency.

Figure 5 shows the results of greedy (G),jump pointer (J) prefetching and those without prefetching (N).The results are normalized to N. Jump-pointer and greedy prefetching improve performance as much as 48% and 18%, respectively. Across all benchmarks, we see improvements of 10% for jump-pointer and 4% for greedy prefetching using the geometric mean.[3].

Effects of Insert and Delete Operations

There are two conflicting factors to consider when examining the performance of insert and delete operations: The increased instruction overhead due to updating of prefetch arrays and jump pointers, and the potential benefit of using prefetching to speed up the search process often employed in the insert/delete function. Only inserts are discussed here, as the overheads of deletes are similar. In mst the node is always inserted at the start of the list, thus we cannot use prefetching to speed-up the insertion. Jump pointer prefetching performs better than PA because there are fewer pointers to update. In health, on the other hand, the node is inserted at the end of the list so prefetching can now be used when traversing the list. The actual instruction overhead is higher for jump pointer prefetching as it uses an expensive modulo operation. With tree add, the performance of the process can also be improved with prefetching and again jump pointer prefetching uses more instructions due to the expensive modulo operator. The instruction overhead of PA is a mere 10 instructions for tree add and the execution time is actually decreased with 10% for PA. Thus, the

instruction overhead of PA and jump pointer prefetching is comparable. PA can speed-up the insertion/deletion with prefetching more effectively than jump pointer prefetching. The performance is worst when the node is inserted first in the LDS.

Table 5: The static instruction and execution time overhead of insert operations in jump pointer prefetching to the left of the slash, and the software version of PA to the right, for three of the kernels.

Kernel	Instruction Overhead	Execution Time Overhead
mst	16/37	11%/25%
health	38/20	-25%/-38%
treeadd	39/10	4%/-10%

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.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions

REFERENCES

1. Magnus Karlsson, Fredrik Dahlgreny, and Per Stenstrom.A Prefetching Technique for Irregular Accesses to Linked Data Structures.
2. Amir Roth and Gurindar S. Sohi**.**Effective Jump-Pointer Prefetching for Linked Data Structures.
3. Brendon Cahoon and Kathryn S. McKinley.Data Flow Analysis for Software Prefetching Linked Data Structures in Java.
4. Steven P.VanderWiel and David J. Lilja. When Caches Aren't Enough:Data PrefetchingTechniques.
5. Chi-Hung Chi, Chi-Sum Ho, Siu-Chung Lau.Reducing Memory Latency Using a Small Software Driven Array Cache.*In Proceedings* of *the 28th Annual Hawaii International Conference on System Sciences* (1995).
6. Lizy Kurian John and Vinod Reddy.Paul T. Hulina and Lee D. Coraor.A Comparative Evaluation of Software Techniques to Hide Memory Latency. *In Proceedings of the 28th Annual Hawaii International Conference on System Sciences* (1995).
7. Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes.
8. Kathryn S McKinley,Brendon Cahoon, Eliot Moss and Darko Stefanovic.Improving Memory Performance for Java.
9. D.C. Burger and T.M. Austin. The SimpleScalar Tool Set,Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, (1997).
10. M. Beckerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport,A. Yoaz, and U.Weiser. Correlated Load-Address Predictors. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 54–63, May 1999.
11. S. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (1996).
12. B. Calder, C. Krintz, S. John, and T.M. Austin. Cache Concious Data Placement. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, 139–149, (1998).