A Prefetching Technique for Irregular Accesses to Linked Data Structures

Magnus Karlsson, Fredrik Dahlgren[†], and Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*karlsson,dahlgren,pers*}*@ce.chalmers.se*

[†]Ericsson Mobile Communications AB
Mobile Phones and Terminals
SE-221 83 Lund, Sweden
fredrik.dahlgren@ecs.ericsson.se

Abstract

Prefetching offers the potential to improve the performance of linked data structure (LDS) traversals. However, previously proposed prefetching methods only work well when there is enough work processing a node that the prefetch latency can be hidden, or when the LDS is long enough and the traversal path is known a priori. This paper presents a prefetching technique called prefetch arrays *which can prefetch both short LDS, as the lists found in hash tables, and trees when the traversal path is not known a priori. We offer two implementations, one software-only and one which combines software annotations with a prefetch engine in hardware. On a pointer-intensive benchmark suite, we show that our implementations reduce the memory stall time by 23% to 51% for the kernels with linked lists, while the other prefetching methods cause reductions that are substantially less. For binary-trees, our hardware method manages to cut nearly 60% of the memory stall time even when the traversal path is not known a priori. However, when the branching factor of the tree is too high, our technique does not improve performance. Another contribution of the paper is that we quantify pointer-chasing found in interesting applications such as OLTP, Expert Systems, DSS, and JAVA codes and discuss which prefetching techniques are relevant to use in each case.*

1 Introduction

Commercial applications, such as database engines, often use hash-tables and trees to represent and store data. These structures, referred to as *linked data structures* (LDS), are often traversed in loops or by recursion. The problem with LDS is the chains of loads that are data dependent of each other that constitute the links of the LDS. These loads can severely limit parallelism if the data is not found in the cache. This is referred to as the pointer-chasing problem. While, prefetching can potentially hide the load latencies of LDS traversals, conventional prefetching techniques fare limited success, because the addresses generated by LDS traversals are often hard to predict a priori.

Recently, five prefetching techniques especially designed to deal with pointer-chasing have been proposed. Three techniques [10, 16, 12] try to overlap the latency of the prefetch that fetches the next node in the LDS, with all the work between two consecutive LDS accesses. When there is not enough work to hide the memory latency, these three methods become less effective. Two other techniques [10, 17] add pointers between non-successive nodes that are used to launch prefetches. While they can potentially hide the load latencies, even when there is little work in each iteration, these techniques require that the traversal path is known a priori and that a large number of nodes are traversed to be effective. Unfortunately as we show in this paper, for important real-world applications, such as database servers, there is little work in each iteration and the traversal path is not known a priori. Note also that balanced hash tables usually have short linked lists with little work in each iteration and thus these methods are not efficient.

In this paper, we present a prefetching technique for hiding the load latencies of tree and list traversals that unlike previous techniques also is effective both when the traversal path is not known a priori and when there is little computation for each node. It accomplishes this with the use of *jump pointers* and *prefetch arrays*. Jump pointers point to nodes located a number of nodes down 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. Prefetch arrays are also used to prefetch the first few nodes in a linked list that do not have any jump pointers referring to them, thus our method is also effective for the short lists found in hash tables.

Our first contribution is an extension of the work of Luk and Mowry [10] by proposing a generalization of the combination of greedy prefetching and jump pointer prefetching. We greedily prefetch several nodes located a number of hops ahead with the use of jump pointers and prefetch arrays that refer to these nodes. Two implementations are proposed and evaluated on a detailed simulation model running pointer-intensive kernels from the Olden benchmark suite and on one kernel extracted from a database server.

The first software-only implementation improves the performance of pointer-intensive kernels by up to 48%. Our method manages to fully hide the latency of 75% of the loads for some kernels. A factor that limits the performance gains of our approach for trees, is the execution overheads caused by the instructions that issue the prefetches of the prefetch arrays. To limit these overheads, we introduce a hardware prefetch engine that issues all the prefetches in the prefetch array with a minimum of instruction overhead. With this hardware, our method outperforms all other proposed prefetching methods for five of the six kernels.

Another contribution of this paper is that we quantify the performance impact of pointer-chasing in what we believe are important application areas. Our measurements show that over 35% of the cache misses of the on-line transaction processing (OLTP) benchmark TPC-B executing on the database server MySQL [19], can be attributed to pointer-chasing. These misses stem from hash tables used to store locks, different buffers for indexes and database entries, and also index tree structures used to find the right entry in the database table. Another application where pointer-chasing can be a performance problem is in expert systems. We also present measurements for five other applications.

This paper continues in Section 2 by introducing an application model to reason about the effectiveness of previous software prefetching techniques for pointer-chasing. In Section 3 our two approaches are presented and then evaluated in Section 5 using the methodology presented in Section 4. Section 6 deals with pointer-chasing in some larger applications, while the related work is presented in Section 7 and the paper is finally concluded in Section 8.

2 Background

To be able to reason about when one prefetching technique is expected to be effective, we present an application model of a traversal of an LDS in Section 2.1. With this model, we show in Section 2.2 when existing pointer-chase prefetching techniques succeed and when they fail.

2.1 The Application Model

A typical LDS traversal algorithm consists of a loop (or a recursion) where a node in the LDS is fetched and some work is performed using the data found in the node. An example code of this is found in Figure 2.1, where a tree is traversed depth-first until a leaf node is found. First, in each iteration some computation is performed using the data found in the node, then the next node is fetched, which in the example is one out of two possible nodes. 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, we are going to denote each of these loads as *pointer-chase loads*.

We will now show that there are four application parameters that turn out to have a significant impact on the efficiency of prefetch techniques: (1) the time to perform the whole loop body, in our model denoted $Work$; (2) the branching factor ($BranchF$) of the LDS (a list has a branching factor of one, a binary-tree two, and so on); (3) the number of nodes traversed, i.e. the *chain length* denoted $ChainL$, and finally, the latency of a load or prefetch denoted $Latency$. All these parameters are shown in Table 1.

```
while (ptr != NULL){ (1) The time to perform one loop body,
  work(ptr);                  denoted Work.
  if (take_left(ptr→data) == TRUE) (2) The branching factor,
    ptr = ptr→left_node;                 BranchF. Here 2.
  else
    ptr = ptr→right_node;
} (3) The number of nodes traversed, denoted ChainL.
```

Figure 1: The pseudo-code of the application model for a binary tree traversal. The model is also applicable when using recursion.

Table 1: The application parameters used in this study.

Parameter	Description
$Work$	The time to perform the whole loop body
$BranchF$	The branching factor
$ChainL$	Number of nodes traversed, the *chain length*
$Latency$	The latency of a load or prefetch operation

The metric we will use to measure the effectiveness of a prefetching technique is the *latency hiding capability* denoted LHC, which is the fraction of the pointer-chase load latency that is hidden by prefetching. In the next section we are going to describe all previously published software prefetching schemes for LDS and use these application parameters to reason about when they are expected to be effective using LHC as the metric.

2.2 Prefetching Techniques for Pointer-Chasing

One of the first software techniques targeting LDS is Luk and Mowry's *greedy prefetching* [10]. It prefetches the next node/nodes in an LDS at the start of each iteration according to Figure 2. For the prefetch to be fully effective, $Work \geq Latency$ 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 reflected in the equations below.

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

Note that these equations are under the assumption that the LDS is a list, or a tree that is traversed depth-first until a leaf node is found and that each pointer-chase load misses in the cache. If a tree is traversed breadth-first, the LHC would be higher than if it is traversed depth-first until a leaf node is found. Note also, that the LHC is the fraction of the pointer-chase load latency that is hidden by prefetching, so it does not take into account any instruction overhead caused by the prefetching technique in question.

To be able to fully hide the pointer-chase load latencies when $Work < Latency$, a prefetch must be issued more than one iteration in advance. This number is called the *prefetch distance* denoted $PrefD$. For a prefetch to be fully effective, it has to be issued sufficiently far in advance to hide the latency of it. This can be calculated as $PrefD = \lceil \frac{Latency}{Work} \rceil$ according to [10]. However, to prefetch one or more nodes $PrefD$ iterations in advance, extra pointers in the code that point to the nodes $PrefD$ iterations ahead, need to be introduced, because originally there are only pointers to the next nodes in the LDS. These extra pointers are called *jump pointers*.

The first scheme to utilize these is *jump pointer prefetching* [10] where a jump pointer was inserted in each node. The jump pointer is initialized to point to a node a number of iterations ahead. As depicted in Figure 2, this pointer is then used to launch a prefetch in each iteration for a node located a number of hops ahead. The first drawback with this solution is that there are no prefetches launched for the first $PrefD - 1$ nodes as there are no jump pointers pointing to these nodes. Thus if $ChainL < PrefD$ the prefetch hiding capability will be zero. If instead $ChainL \geq PrefD$, ignoring the effect on the LHC of the first $PrefD - 1$ load misses, and $BranchF = 1$ there is only one possible traversal path, thus $LHC = 1$ if $PrefD$ is set properly. However, if $BranchF > 1$ and we assume that each node will be visited with equal probability and that the tree is traversed depth-first until the first leaf node is reached, the probability that the correct node will be prefetched is only $1/BranchF^{PrefD}$, thus $LHC = 1/BranchF^{PrefD}$. The effectiveness is thus as follows:

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

If the traversal path is known beforehand, the fraction in the above expression can be turned into a 1, by using this knowledge to initialize the jump pointers to point down the correct path.

Chain jumping [17] is a variation on jump pointer prefetching that can only be used when the LDS node also contains a pointer to some data that is used in each iteration. For a prefetch approach to be fully effective, this data must also be prefetched along with the node. Jump pointer prefetching solves this by storing two jump pointers, one to the node and one to the corresponding data and launches prefetches for both. Chain jumping saves the storage of the last jump pointer by launching that prefetch last in the iteration instead, using the pointer prefetched at the beginning of the iteration. This approach will only work well when $Work \sim Latency$, as there must be enough computation between the first prefetch in the iteration and the last one, using the pointer fetched by the first prefetch. We will only look at jump pointer prefetching in this paper, as chain jumping only can be used for one application that we study in Section 5, and for that one it was proved in [17] only to be marginally more effective than basic jump pointer prefetching.

Overall, the techniques presented so far have two major drawbacks: they are not effective either when the chain length and the amount of computation per iteration are small or when the amount of computation per iteration is small and the traversal path is not known a priori. We will show that our techniques, presented in the next section, are expected to be effective also under these conditions when the source of the pointer-chasing is a loop or a recursion. This is common in many important applications as we will see in Section 6.

3 Our Prefetching Techniques

In this section we describe our prefetching techniques for LDS. Section 3.1 describes our software method and Section 3.2 describes our proposed block-prefetch instruction and how it is used in an LDS application. For all the prefetching methods, we will use the application model presented in Section 2 to describe when the techniques are effective and when they are not.

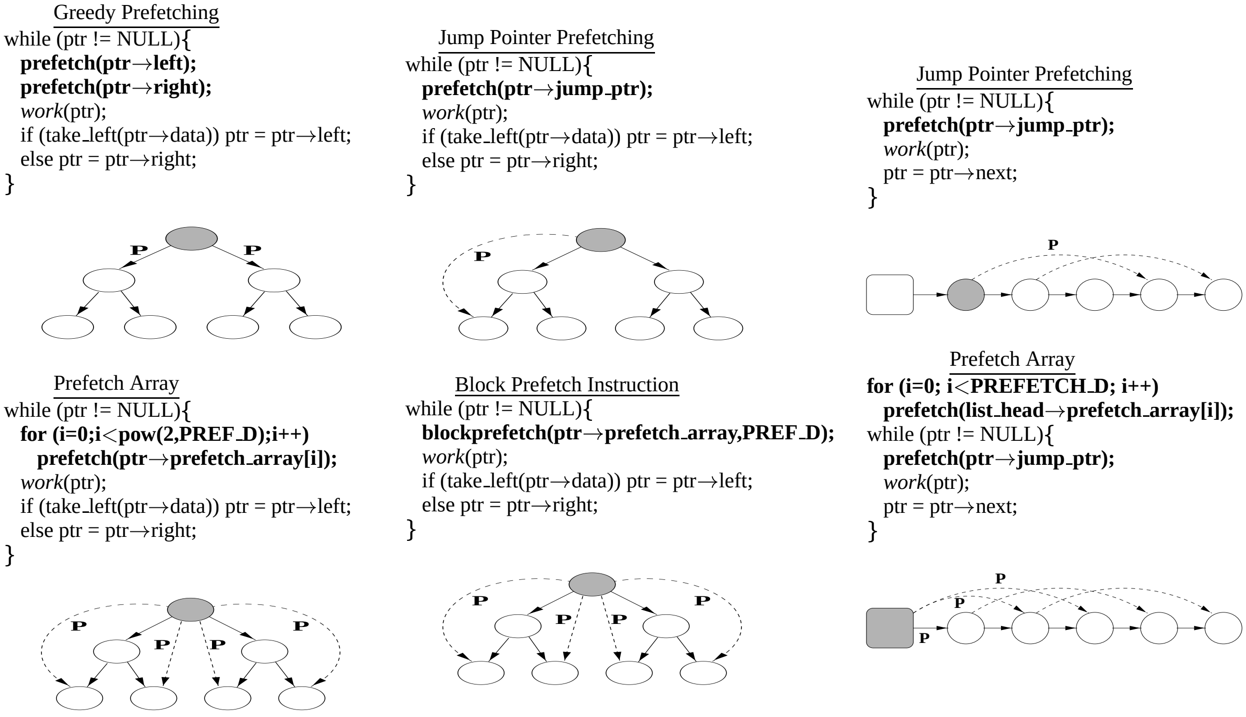
3.1 Prefetch Arrays: A Software Approach

Our prefetching technique is called the *prefetch arrays technique (PA)* aims at hiding the memory latency of LDS also when the other methods fail. Our method makes this possible by trading off memory overhead and bandwidth for fewer misses. We start by describing the basic mechanism in Section 3.1.1 and end it by discussing its properties in Section 3.1.2.

3.1.1 Prefetch Approach

The main idea behind our approach is to aggressively prefetch all possible nodes a number of iterations ahead equal to the prefetch distance. To be able to do this, we associate a number of jump pointers with each node. These jump pointers are allocated in consecutive memory locations, in what we call a *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 2.

When $BranchF = 1$, jump pointer prefetching has the disadvantage that it does not prefetch the first $PrefD - 1$ nodes as shown in Figure 2. This is very ineffective for short lists, as those found in hash tables for example. With our method, we instead create a prefetch array at the head of the list that points to the first $PrefD - 1$ nodes that the regular jump pointers found in the nodes 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



Greedy Prefetching

```
while (ptr != NULL){
  prefetch(ptr→left);
  prefetch(ptr→right);
  work(ptr);
  if (take_left(ptr→data)) ptr = ptr→left;
  else ptr = ptr→right;
}
```

Jump Pointer Prefetching

```
while (ptr != NULL){
  prefetch(ptr→jump_ptr);
  work(ptr);
  if (take_left(ptr→data)) ptr = ptr→left;
  else ptr = ptr→right;
}
```

Jump Pointer Prefetching

```
while (ptr != NULL){
  prefetch(ptr→jump_ptr);
  work(ptr);
  ptr = ptr→next;
}
```

Prefetch Array

```
while (ptr != NULL){
  for (i=0;i<pow(2,PREF_D);i++)
    prefetch(ptr→prefetch_array[i]);
  work(ptr);
  if (take_left(ptr→data)) ptr = ptr→left;
  else ptr = ptr→right;
}
```

Block Prefetch Instruction

```
while (ptr != NULL){
  blockprefetch(ptr→prefetch_array,PREF_D);
  work(ptr);
  if (take_left(ptr→data)) ptr = ptr→left;
  else ptr = ptr→right;
}
```

Prefetch Array

```
for (i=0; i<PREFETCH_D; i++)
  prefetch(list_head→prefetch_array[i]);
while (ptr != NULL){
  prefetch(ptr→jump_ptr);
  work(ptr);
  ptr = ptr→next;
}
```

Figure 2: The implementation of the software approaches in an example of a tree and a list traversal. Jump pointers are dashed and a 'P' designates that the pointer is used for launching prefetches. The pow-function in one code example calculates 2^{PREF_D}.

same as for the binary-tree. In the next section, we will start by deriving the LHC of our prefetching method.

3.1.2 Discussion

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 $ChainL \geq PrefD$ (ignoring the misses of the first $PrefD - 1$ nodes). However, if $ChainL < PrefD$ there will be no prefetches:

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

For a list, the effectiveness will also be 1 when $ChainL \geq PrefD$ 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 $\min(1, \frac{Work}{Latency})$. The prefetches for the following nodes will by that time have arrived, and the LHC of those will be one. The equations are summarized below.

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

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 $PrefD = 1$ we get Luk and Mowry's greedy prefetching, and we do not need any prefetch arrays as the pointers to the next nodes already exist in the current node. 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 PA to be effective.

How bandwidth limitations affect our prefetching method will be examined in Section 5.3 and insert/delete instruction overhead is examined in Section 5.4. In the next section we will introduce a new instruction and a prefetch engine that eliminates most of the instruction overhead as-

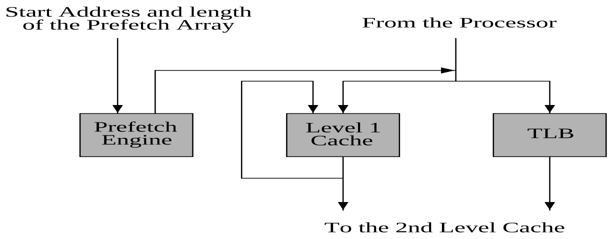


Figure 3: The location of the prefetch engine for an architecture where the physical or virtual L1 cache is accessed at the same time as the TLB. If the TLB needs to accessed before the physical cache, all the accesses of the prefetch engine needs to pass through the TLB first also.

sociated with launching prefetches using the prefetch array.

3.2 Prefetch Arrays: A Hardware Approach

To alleviate the potential problem of high instruction overhead when prefetching using the prefetch arrays, we propose a new instruction called a *block prefetch operation*. This instruction takes two arguments: a pointer to the beginning of the prefetch array, and the length of the prefetch array. When executed it triggers a prefetch engine that starts to launch prefetches for the addresses located in the prefetch array. When it reaches the end of the prefetch array, it stops. Note in Figure 2 that it substitutes the for-loop of the all-software approach with just one instruction. This instruction should increase performance substantially, especially when $BranchF > 1$ and $Work$ is small. For all other cases the performance improvements should be quite small. Examples of other prefetch engines can be found in [18, 3, 12, 16, 17]. The first two engines prefetch single addresses and addresses with strides and all initial addresses are produced by a compiler. The remaining three prefetch engines will be discussed in Section 7 as they also target some aspects of pointer-chasing.

This simple prefetch engine can be implemented alongside the first level cache according to Figure 3. The figure depicts a virtually-addressed cache. If the first level cache is physically addressed, all cache look-ups from the prefetch engine has to pass through the TLB first. When the engine gets the first address of the prefetch array and the length of it from the processor, it reads the first entry of the prefetch array from the first level cache. It then launches a prefetch for this address. The prefetch engine then repeats this procedure for the next word in the prefetch array. This process is repeated a number of times equal to the prefetch array length. Should a cache miss occur, the cache loads the cache block and the prefetch engine launches the prefetch as soon as the cache block arrives. If there is a page fault, the prefetch engine aborts that prefetch.

4 Methodology and Benchmarks

In this section, we present our uniprocessor simulation model and the two sets of benchmarks that are used. The first set of *kernel* benchmarks is taken from the Olden benchmark suite [15] with the addition of a kernel taken from the database server MySQL [19]. All our simulation results were obtained using the execution-driven simulator SimICS [11] that models a SPARCv8 architecture. The applications are executed on Linux which is simulated on top of SimICS. However, for the kernels we do not account for operating system effects.

The baseline architectural parameters can be found in Table 2. Note that only data passes through the memory hierarchy. Instructions are always delivered to the processor in a single cycle. How the results of this paper extends to systems with other features such as super-scalar and out-of-order processors, copy-back caches, and bandwidth limitations will be discussed where appropriate in Section 5.

The kernels we have used are presented in Table 3. The kernels are chosen to represent a broad spectrum of values regarding $ChainL$ and $Work$. Note that the kernels typically have a instruction footprint that is less than 16KB, which means that a moderately-sized instruction cache would remove most of the instruction stall time for these kernels.

`DB.tree` is a kernel taken from the database server MySQL, that traverses an index tree, something that is performed for each transaction that arrives to an OLTP-system, for example. The traversal is performed one node per level until a leaf node is found. After that another traversal (belonging to another transaction) is performed that may take another path down the tree.

The other benchmarks are taken from the Olden benchmarks suite [15], except for `mst.long` which is `mst` with longer LDS chains. This was done by dividing the size of the hash-table by four. We included this kernel because there was no other program in the suite that had long static lists with little computation performed for each node. All programs were compiled with egcs 1.1.1 with optimization flag -O3 and loop unrolling was applied to the for-loop that launches the prefetches of the prefetch array in the all software approach. We do not gather statistics for the initialization/allocation phase of the benchmark, as we will study the effects of insertion/deletion of nodes separately in Section 5.4. Note that for `health`, inserts and deletes occur frequently during the execution so they are included in the statistics. We have also used complete applications that will be presented in Section 6.

5 Experimental Results

Before we take a look at the impact of pointer-chasing and the prefetching techniques in this paper for large applications, we will verify that the techniques behave as expected by studying them on the simple Olden kernels in Section 5.1. To start with, the prefetch distance has been fixed at 3 for all kernels, except PA for trees where it is set to 2. How effective the techniques are when the memory latency and prefetch distance is varied, is studied in Section 5.2. Initially we assume a contention free memory system. Bandwidth limitations is something that can limit the improvements of prefetching, and this is studied in Section

Table 2: The baseline architectural parameters used in this study. Note that some of these parameters are changed in later sections.

Processor Core	Single-issue, in-order, 500 MHz SPARC processor with blocking loads.
L1 Data Cache	64KB, 64B line, 4-way associative, write-through, 1 cycle access. 16 outstanding writes allowed.
L2 Data Cache	512KB, 64B line, 4-way associative, write-through, 20 cycle access. 16 outstanding writes allowed.
Memory	100 cycle memory latency. Contention free.
Prefetches	Non-binding, aborts on page faults. Maximum of 8 outstanding prefetches allowed. Prefetch engine can launch a prefetch every other cycle, if the prefetch array is cached.

Table 3: The kernels used in the study. The first three traverse lists, the two following traverse binary trees, and the last one a quad-tree.

Kernel	Input Param.	LDS Type	$ChainL$	$Work$
mst	1024 nodes	static hash tables	2-4	little
mst.long	1024 nodes	static hash tables	8-16	little
health	5 levels, 500 iter.	dynamic lists	1-200	medium
DB.tree	100000 nodes	dynamic binary-tree, top-down traversal	16	medium
treeadd	1M nodes	static binary-tree, every node visited, traversal order known	19	little
perimeter	2K x 2K image	static quad-tree, every node visited, traversal order known	11	medium

5.3. Finally the costs of updating the prefetch arrays and the jump pointers are studied in Section 5.4.

5.1 Tree and List Traversal Results

The performance of a prefetching technique for any system can roughly be gauged by measuring four metrics: execution time, full coverage, partial coverage, and efficiency. Full coverage is the ratio of cache misses eliminated by the prefetching method to the total number of cache misses. Partial coverage is the ratio of cache misses to blocks for which prefetches have been launched but have not been completed to the total number of misses. Efficiency is the fraction of all prefetched cache blocks that are later accessed by an instruction. We will start by discussing the effect of prefetching on the execution time of list and tree traversals. Figure 4 shows the normalized execution times of the kernels with different prefetching approaches and without them. The execution time is normalized to the base case without prefetching. Each bar is subdivided into busy time, instruction overhead due to the prefetching technique, and memory stall time. Note that there is no contention on the bus to be able to disregard effects stemming from limited bandwidth.

`Mst` is a hash-table benchmark where $ChainL$ is between two and four and $Work \ll Latency$. This means that greedy prefetching is not expected to be effective as $Work$ is low. As expected, it only improves performance by 2%. Jump pointer prefetching is also ineffective as $ChainL < PrefD$ most of the time. It only improves performance by 1%. However, with PA, part of the load latency of the first nodes in the list will also be hidden and thus we get a performance improvement even for short lists. The software approach improves performance by 20%. The hardware approach is only marginally better than the pure software one and improves performance by 22%. This is due to the fact that the instruction overhead of the software approach is only four instructions, because the prefetch array contains only two jump pointers, while the hardware approach has an overhead of two instructions. The instruction overhead of the hardware PA is slightly higher than for jump pointer prefetching and this is because the block prefetch instruction takes two arguments instead of one. This extra instruction overhead can also be observed for `mst.long` and `treeadd`, but for `health` and `DB.tree` this overhead is dwarfed by other instruction overheads of jump pointer prefetching as will be discussed later.

Figure 5 shows the coverage of the kernels divided into full coverage and partial coverage. For `mst`, we see that jump pointer prefetching prefetches the nodes early enough but very few of them, while the prefetch array technique prefetches more nodes early enough to hide a larger part of the memory latency than jump pointer prefetching. The reason why the coverage is not 100% for greedy prefetching and PA is the following. The pointers used to launch the prefetches only point to the first word of each node. All the techniques thus only launch a prefetch for the cache block that the first word of the node is located on, and if the node is located across two cache blocks the second block will thus not be prefetched. So even though greedy prefetching and PA launch prefetches for the first word of all nodes, there are cache misses for the nodes located across two cache blocks. Adding jump pointers to nodes increases the number of nodes located across cache blocks as can bee seen when comparing the coverage of greedy prefetching with PA.

Table 4 shows the efficiency of the prefetching techniques. Greedy prefetching has an efficiency of 100%, because it only prefetches the next node in the list. Jump pointer prefetching has only a 42% efficiency because sometimes the right node is found earlier in the list than the fourth node in the list that jump pointer prefetching prefetches. PA has an efficiency of 75%.

The kernel `mst.long` has a large $ChainL$ otherwise it

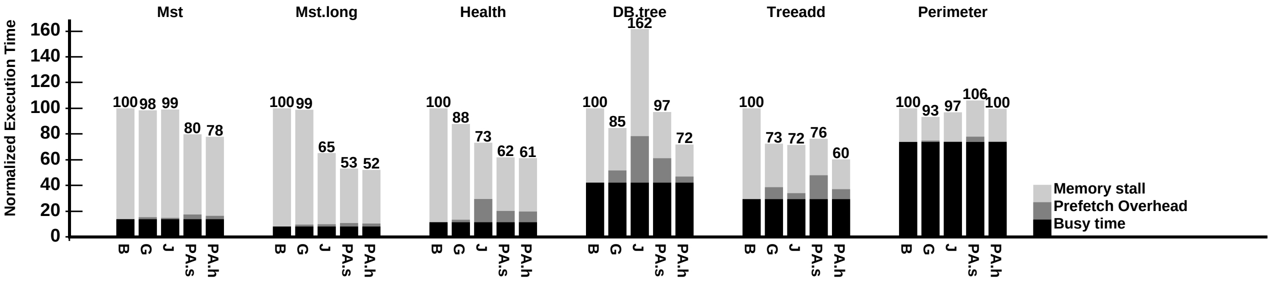


Figure 4: The execution time of the benchmarks normalized to the execution time without prefetching. In the figure, ’B’ is the base case without prefetching, ’G’ is greedy prefetching, ’J’ is jump pointer prefetching, ’PA.s’ is the software implementation of prefetch arrays, and ’PA.h’ is the hardware implementation.

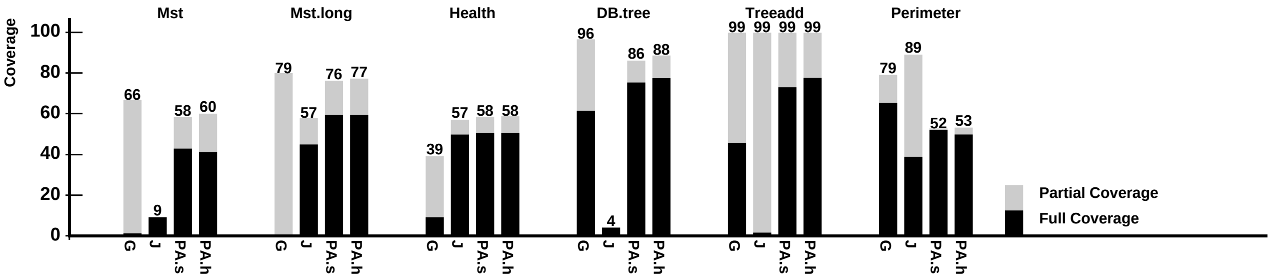


Figure 5: The coverage subdivided into full and partial coverage of the benchmarks. In the figure, ’G’ is greedy prefetching, ’J’ is jump pointer prefetching, ’PA.s’ is the software implementation of prefetch arrays, and ’PA.h’ is the hardware implementation.

Table 4: The efficiency of the prefetching schemes for all the benchmarks.

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%

is the same as `mst`. This means that jump pointer prefetching is now expected to be more effective as we can see from the much higher coverage and efficiency values. It improves performance by 35%. However, PA is still a better approach as it improves performance by 47% and 48%, respectively.

`Health` differs from the previous hash table kernels in three aspects: there are no hash-tables, $Work$ is not as small, and LDS nodes are inserted and deleted frequently during the execution. The last fact should indicate that jump pointer prefetching and prefetch arrays should be less effective as they incur overheads for insert and delete operations. However all of the aforementioned approaches are more effective than greedy prefetching. Jump pointer prefetching improves the performance by 27%, and PA with 38% and 39%, respectively. The reason that PA is better than jump pointer prefetching is that the insert and delete operations are less costly for PA as will be explained in Section 5.4. The memory stall time reduction is comparable for both the techniques as $ChainL$ can be up to 200.

`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. As can be seen in Figures 4 and 5, jump pointer prefetching is not a useful alternative for applications where the traversal path 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 $Latency$.

`Treeadd` differs from the previous kernel in that the traversal path is known a priori. For this kernel jump pointer prefetching provides a performance increase as seen in Figure 4. Note that 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 varying 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 quad-tree 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.

The question now is how a multiple-issue processor would affect the memory stall time and busy time. As far as memory-stall time, we do not expect it to be decreased because most loads in the kernels are data dependent. As a result there is little opportunity for overlapping cache misses. Regarding the busy time, it will remove some of the instruction overhead associated with the prefetching techniques. If a processor, for example, has four load/prefetch functional units there would be no need for a special prefetch engine for binary-trees as the prefetches could all be launched simultaneously by the four functional units. Another simplification in our modeled architecture is that we have write-through caches. However, the results would be similar with copy-back caches because a vast majority of the accesses (typically $> 98\%$) are reads in these kernels.

To summarize, PA work well for hash-tables, all sorts of lists and binary trees with an a priori unknown traversal path. If the processor has not enough load units, our hardware prefetch engine must be used for good performance for trees. However, when the traversal path in a tree is known, jump pointer prefetching can provide better performance improvements than PA. For quad trees, and trees with a higher $BranchF$, with an a priori unknown traversal path, no known prefetching technique works well when $Work \ll Latency$.

5.2 Impact of Memory Latency

Figure 6 shows the normalized execution times of the kernels with and without the prefetching techniques when the memory latency is changed from 100 cycles to 200 cycles and for two different prefetch distances. The kernel `perimeter` was not included because the results are similar to `treeadd`. For the results with the increased prefetch distance, the maximum number of outstanding prefetches is raised to 24 for the tree kernels, as the number of maximum outstanding prefetches should be larger than or equal to $PrefD \cdot PrefD^{BranchF}$ for maximum performance gain with PA.

There are two main observations regarding the list kernels. First, all prefetching techniques manage to hide nearly as much latency or more as when the memory latency was lower. Second, there is not much reason to increase the prefetch distance. Jump pointer prefetching fails to prefetch the first few nodes if $PrefD$ is too high as can be seen in `mst`. However, it can be slightly beneficial for longer lists as in `health`. PA are more robust, as increasing the prefetch distance will not make it miss the first $PrefD - 1$ nodes.

For the binary-tree kernels the results are different. Increasing $PrefD$ for jump pointer prefetching enables it to hide more latency as long as the binary tree is large enough, as the instruction overhead is constant regardless of $PrefD$. PA on the other hand, will have to launch more prefetches if the prefetch distance is increased. For the software solution this also means more instruction overhead. The effect of this can especially be seen in `DB.tree`, where the performance is increased by 2% for jump pointer prefetching when $PrefD$ is increased, while the performance of PA is degraded by 70% and 32%, compared to the base case. We conjectured that the performance degradation was attributed to cache pollution. In order to test this hypothesis, we reduced the size of the caches for the base system and the system with our prefetch technique. While the miss rate only changed marginally for the system with no prefetch, a significant increase was noted for the system with prefetches. This indicates that the reason for the poor performance is mainly due cache pollution. Note that the hardware implementation of PA is still the best prefetch technique for 4 out of the 5 kernels.

To summarize this section, all prefetching techniques are robust when it comes to dealing with increased memory latency. For jump pointer prefetching, $PrefD$ can be increased to compensate for the increase in memory latency when traversing trees, but not for short lists. However, PA needs a low $PrefD$ to be effective for trees. In the next section bandwidth limitations and its effect on the prefetching techniques will be evaluated.

5.3 Effects of Limited Bandwidth

Bandwidth limitations can seriously degrade the performance improvements of the techniques studied in this paper. In this section, two systems are studied. The first one represents a high-end system with a memory bus bandwidth of 2.4 GBytes/s, and the second one represents a low-end system with a bandwidth of 640 MBytes/s.

Figure 7 shows the normalized execution times of the kernels for the high-end system. From the figure we can see that there is ample of bandwidth for all the prefetching techniques. PA is the technique that issues the most prefetches. However, even for this technique implemented in the tree kernels, there is enough bandwidth to accommodate most of the extra traffic caused by the prefetches.

Going instead to the low-end system, we can see that the potential performance improvements seen in Section 5.1 do not show up to the same extent in Figure 8. For lists, PA is still the best technique, but the performance improvements of `mst` have been cut from 20% and 22% to 6% and 7%. For `mst.long` the degradation is from 47% and 48% to 29% for both, and for `health` it decreases from 38% and 39% to 18% for both techniques. The reason that PA is still a viable technique for list traversals on low-end systems, is that it does not produce many useless prefetches when $BranchF = 1$.

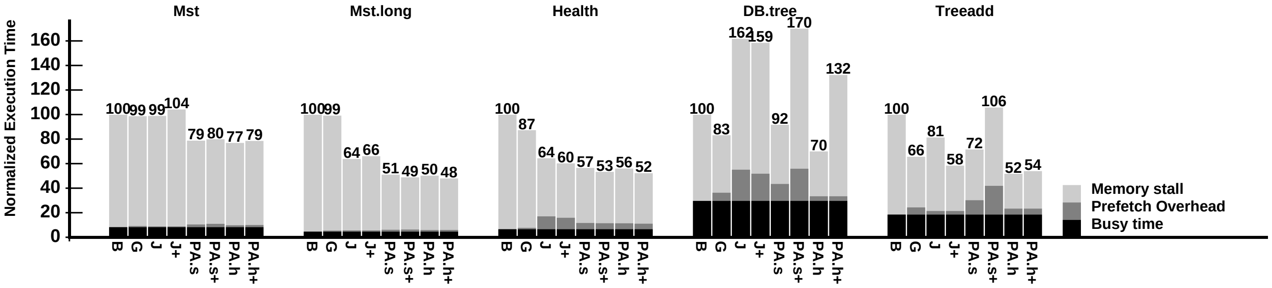


Figure 6: The execution time when the memory latency is 200 cycles normalized to the execution time without prefetching. In the figure, 'B' is the base case without prefetching, 'G' is greedy prefetching, 'J' is jump pointer prefetching, 'PA.s' is the software implementation of prefetch arrays, and 'PA.h' is the hardware implementation. A '+' means that the prefetch distance has been increased by one.

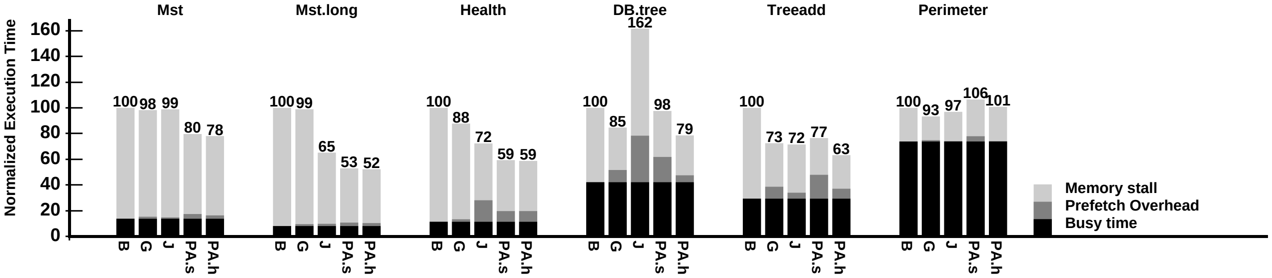


Figure 7: The execution time for a high bandwidth system, normalized to the execution time without prefetching.

When $BranchF > 1$, PA starts to generate many useless prefetches that can cause contention in the system. This is what we can see for the tree kernels in Figure 8. Here, PA does not offer a performance improvement but for `treeadd` and hardware prefetch arrays where there is a 5% improvement. When the traversal path is not known a priori, as in `DB.tree`, the only viable approach is greedy prefetching that still can offer a performance improvement. For the kernels `treeadd` and `perimeter` the best approach should be jump pointer prefetching after an adjustment of the prefetch distance.

To summarize, PA offers the best performance improvement for high-end systems and for lists and hash-tables on low-end systems. However, for trees on low-end systems the bandwidth is not high enough to accommodate the aggressive prefetching.

5.4 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. We will only discuss inserts, as the overheads of deletes are similar.

The average static instruction overheads for inserting a node are shown in Table 5 for three kernels. 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 less 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. As seen in Table 5, we actually get a performance increase. The actual instruction overhead is higher for jump pointer prefetching as it uses an expensive modulo operation. With `treeadd`, 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 `treeadd` and the execution time is actually decreased with 10% for PA.

To summarize, 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%

6 Pointer-Chasing in some Applications

To examine the characteristics and performance impact of pointer-chasing for emerging application domains, we

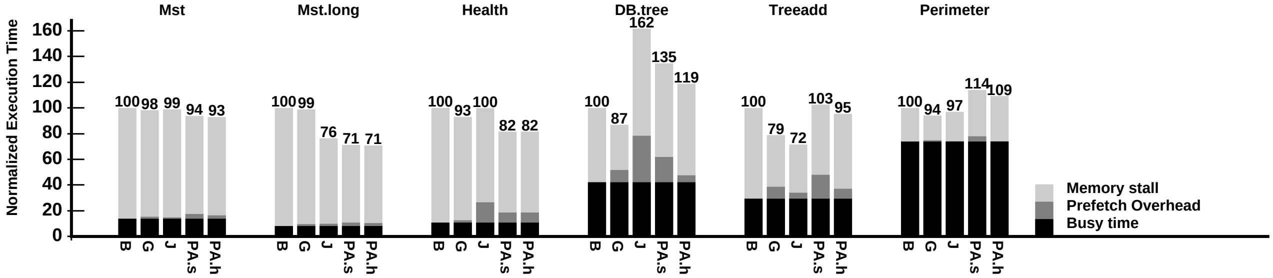


Figure 8: The execution time for a low bandwidth system, normalized to the execution time without prefetching.

have measured the critical application parameters for eight applications using SimICS. In our model, a load is a pointer-chasing load if the address it uses was loaded into the register file by a previous load. This address may have been modified by a number of arithmetic operations since it was loaded and also moved inside the register file, but it is still a pointer-chasing load in our model. The two loads in the above example constitute a *pointer-chasing chain (PCC)*. We say that a PCC is ended if there are no new loads dependent on the last pointer-chasing load in that chain 10000 instructions after this last load was executed. At that time we increment the number of pointer-chasing loads in the execution by the number of pointer-chasing loads in the ended chain, and mark that we have had one more PCC with a $ChainL$ equal to the number of pointer-chasing loads in the chain. If there are more than one load dependent on the same pointer-chasing load, we only include the load that will give rise to the longest PCC. Finally, $Work$ is the number of instructions in between two pointer-chasing loads.

The first application studied is the on-line transaction processing (OLTP) benchmark TPC-B [6] with 40 branches executing 5000 transactions on the database server MySQL [19], and the second one is the decision support benchmark (DSS) TPC-D's [7] query Q3 executing on a 200 MB database with full indexes. TPC-D is also running on MySQL. JVM is represented by the JAVA JIT-compiler Kaffe [20] running the SPECJVM98 program "jess". We have run four more programs on Kaffe ranging from linpack to a raytracer, however we arrive to the same conclusions for all the JAVA programs. The fourth application is a commercial expert system that interprets network traffic data in search of possible network security breaches or attempts. 3D-games are represented by Quake [8]. There is also have a raytracer from the SPLASH-2 application suite [2], an MPEG2 decoder [13] and a handwriting recognition program [14].

Table 6 shows the measured numbers of $ChainL$, $Work$, the percentage of cache misses due to pointer-chasing chain loads, the cache miss ratio and the ratio of pointer-chasing chain loads in the application. The last value is calculated as the number of dynamic PCC loads in the execution divided by the total number of dynamic loads. The numbers are derived assuming a 512 KByte cache. As PCC we only account for chains were $ChainL \geq 2$.

OLTP has a fair amount of PCC that miss in the cache as 35.7% of the cache misses are due to this. The chain length varies, but the average is around 10. $Work$ is most of the time very low, around 10 cycles, and sometimes high, above 1000 cycles. The pointer-chasing misses with a low $Work$ mostly stem from hash-tables used to store and retrieve buffers, locks and other data, and binary-trees used to store the indexes. All the traversal of these structures are input-data dependent as they depend on the actual transaction that arrives to the database server, i.e. jump pointer prefetching cannot be used. PA can be successfully used in this application to improve the performance of both the index tree (the `DB.tree` kernel) and the hash-tables similar to `mst`. Pointer-chasing chains may not seem to be a problem as the cache miss ratio is only 0.9% for a 512K cache on a uniprocessor. However, we conjecture that the ratio of cache misses due to pointer-chasing increases with database size as the index-tree and some of the hash-tables grow with database size, and as the execution time spent traversing these will also grow, the ratio of PCC in the execution will also increase.

DSS does not suffer from nearly any pointer-chasing chain cache misses and there are only 2% PCC in the execution. As in OLTP, the index tree and the hash tables are used to retrieve data from the database. By contrast, often in DSS a large number of database entries are accessed and these are not independent of each other. Thus, in the extreme case, the index tree is accessed once and then transformed into another structure. This structure is then traversed sequentially to retrieve all the database entries and pointer-chasing is thus avoided.

The JAVA application for which we present results for here, does not have much pointer-chasing in itself. Looking at Table 6 we can see that nevertheless 17.2% of the loads constitute PCC but only 7.6% of the cache misses are due to these. However, further studies are needed in order to examine under which circumstances the PCC miss. Most PCC are short but there exist chains of most lengths and $Work$ varies a lot. The PCC in this application does not stem from a low number of sources and from distinct PCs as in OLTP. Instead it is hard to pin-point the sources as they are so many, and the pointer-chase chains are often

Table 6: Measured typical values of the application parameters for some applications. CMR stands for cache miss ratio.

Application	$ChainL$	$Work$	PCC CM / CMR	PCC in the execution	Comments
OLTP	10 varying	10 or $>$1000	35.7%/0.9%	10.9%	Distinct sources
DSS	2	15	$<$0.1%/1.1%	2.0%	
Kaffe	$\leq$3 varying	Any	7.6%/0.7%	17.2%	Non-distinct sources
Exp Sys	2-11	$<$40 or $>$950	9.5%0.06%	16.0%	List Traversals
Exp Sys.state	$\leq$15	$<$10 or $\approx$180	31.9%/13.8%	14.3%	List traversals
Raytrace	$<$10	$<$10 or $>$700	20.9%/0.32%	14.1%	Tree traversals
MPEG	2-5	1000	4.2%/0.10%	0.4%	Low cache miss ratio
Handwriting	$\leq$3	$<$30 or $>$900	7.1%/0.31%	4.2%	Numerical Loops
Quake	$<$10	$<$60 or $>$850	12.2%/0.51%	7.4%	No source code

produced by many different instructions in a complex way. This suggests that all prefetch methods that require annotations to the programs as the ones presented in this paper, would not be a good choice for a JVM.

The cache miss ratio due to PCC in the expert system we ran depends heavily if the rule set saves state (facts) or not. If facts are saved, they are stored in linked data structures, mostly lists. One or more of these are then traversed for every succeeding incoming fact. Without any saved facts the cache miss ratio due to pointer-chasing is 9.5%, however with saved facts it is increased to 31.9%. Considering this and that the cache miss ratio is 13.8% when state is saved, PCC can degrade performance a lot for this application. As the lists can be both long and short and the sources of all the pointer-chasing chain cache misses are few, we believe that our technique could be successfully used to improve the performance of this application.

The raytracer traverses a tree as it traces the ray and the cache miss ratio due to PCC is 20.9% and the PCC loads in the execution is 14.1%, but the cache miss ratio is only 0.32%. We conjecture that the cache miss ratio most of the time will increase as the number of objects in the image is increased. The tree traversals in raytracing either calculate a lot in each node or just quickly goes on to the next. Unfortunately, the tree often has a high branching factor which means that our technique and greedy prefetching cannot be used. Jump pointer prefetching cannot either be used as the traversal path through the tree is not known a priori. For the other three applications the number of cache misses due to PCC were low.

To summarize, pointer-chasing chains can be a problem for the expert system we tried, and we conjecture that it also is a problem for OLTP with large database sizes. We expect that PA can be used successfully for OLTP and the expert system. However for JAVA and raytracing all the techniques evaluated in this paper would not work.

7 Related Work

Roth and Sohi presented in [17] *root jumping*. They add a pointer between the roots of LDS, and the LDS are linked in the order that they will be traversed. When one LDS is traversed, prefetches for the next LDS is launched via the root pointer. This is a good approach as long as there is a fixed order between the LDS traversals. If there is not, it cannot be used. Two other drawbacks are that there cannot be too much work between each LDS traversal and the LDS chains cannot be too long as the nodes would be prefetched to early and evicted from the cache before the next LDS was traversed. This approach could be used in conjunction with all the techniques evaluated in this paper.

Three hardware techniques that launch prefetches for LDS have been proposed earlier. Mehrotra proposed the indirect reference buffer [12], which targets both sequential patterns and pointer-chasing. However, it can only prefetch lists and it is only fully effective when the amount of work in an iteration is larger than the prefetch latency. Dependence-based prefetching (DBP) [16] has the last drawback also, but the prefetching mechanism is not limited to lists. Our technique does not have these drawbacks. In [17], Roth et al. combined jump pointer prefetching with a hardware mechanism based on the DBP. For `health`, the data structures connected to each node in the LDS are prefetched using this hardware, and this improves performance a lot. However, for the other applications the improvements or degradations are small. This technique still needs an a priori known traversal path and the LDS cannot be short. Our technique can handle both these cases.

A novel approach presented by Bekerman et al. in [1] records the LDS traversals in the load address predictor in the processor. When the beginning of one of these load address patterns is later repeated, the processor will speculate that the same pattern will occur again and give the speculated address to each corresponding load. This works well for static lists that are not too short, but not as well when the lists are dynamic or short, or when traversing trees and other LDS.

Cache conscious data placement has been explored in [4, 5, 9]. The technique tries to pack LDS nodes that are accessed in a row into the same or consecutive cache blocks. Prefetching the next few blocks is then simple. This technique can dramatically improve performance for all types of LDS even when there are little work performed in each node. The drawbacks are that the overhead for rearranging the nodes is large, making the technique most suitable for highly static structures, and that the technique requires knowledge of the cache parameters. Luk *et al.* [9] are more aggressive in their relocation, however they require modifications to the hardware.

8 Conclusion

In this paper we propose a new prefetching strategy for linked data structures called prefetch arrays. We propose two implementations, one software-only approach and one with a modest hardware support in shape of a prefetch engine. Our two implementations are then compared to previously proposed prefetching techniques on a simulation model of a uniprocessor system.

Our experiments show that the prefetch arrays technique can effectively prefetch even short lists, as those found in balanced hash tables, something that the other techniques fail doing efficiently. It also provides the best performance improvements for longer lists, however the performance differences to other techniques are less. We also show that our hardware technique can successfully prefetch binary-trees even when the traversal path is not know a priori and when the amount of work in a node is not enough to hide the prefetch latency. However, cache pollution can limit the performance gains when traversing trees with a high branching factor. Our experiments also demonstrate that there seems to be ample bandwidth in today's high-end systems to accommodate our prefetching techniques. However, in low-end systems the memory bus is expected to become a bottleneck when prefetching trees, but not for lists.

A second contribution of this paper is the quantification of pointer-chasing in what we believe are important applications. For these applications we have measured critical application parameters that we then use to argue about the success and failure of different prefetching algorithms. We find that OLTP, a raytracer and an expert system has a fair amount of pointer-chasing that really misses in the cache. We argue that many of the cache misses for OLTP and the expert system can be removed with our prefetching techniques. The JAVA just-in-time compiler we ran had a lot of pointer-chasing, however it did not miss in the cache and thus it was not a problem. Other applications, such as decision-support systems and an MPEG-decoder did not show much pointer-chasing. With this we have shown that there are important application classes that could benefit from the prefetching technique presented in this paper.

Acknowledgments

Appreciations to Jim Nilsson for the original memory hierarchy. This research has been supported by grants and equipment from Sun Microsystems, Swedish Research Council on Engineering Science (TFR) under contract number 311-97-99, and Swedish Council for Planning and Coordination of Research (FRN) under contract number 96238.

References

[1] 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.

[2] S. Cameron Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[3] Tien-Fu Chen. Effective Programmable Prefetch Engine for On-Chip Caches. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 237–242, November 1995.

[4] T. Chilimbi, M. Hill, and J. Larus. Cache-Concious Structure Layout. *SIGPLAN-Notices*, 34(5):1–12, May 1999.

[5] T. Chilimbi and J. Larus. Using Generational Garbage Collection to Implement Cache Concious Data Placement. *SIGPLAN-Notices*, 34(3):37–48, March 1999.

[6] Transaction Processing Performance Council. *TPC Benchmark B (Online Transaction Processing) Standard Specification*, December 1990.

[7] Transaction Processing Performance Council. *TPC Benchmark D (Decision Support) Standard Specification Revision 1.1*, December 1995.

[8] ID Software. http://www.idsoftware.com. *Quake*, October 1997.

[9] C-K. Luk and T. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 88–99, May 1999.

[10] C-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[11] Peter. S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: A Virtual Workstation. In *Proceedings of USENIX'98*, pages 119–130, June 1998.

[12] S. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1996.

[13] MPEG Organization. http://www.mpeg.org. *MPEG2 decoder*, January 1996.

[14] National Institute of Standards and Technology. http://www.nist.gov/itl/div894/894.03/databases/defs/ nist_ocr.html. *Public Domain OCR: NIST Form-Based Handprint Recognition System*, February 1997.

[15] A. Rogers, M Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[16] A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[17] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 111–121, May 1999.

[18] J. Skeppstedt and M. Dubois. Hybrid Compiler/Hardware Prefetching for Multiprocessors Using Low-Overhead Cache miss Traps. In *Proceedings of the International Conference on Parallel Processing*, pages 298–305, September 1997.

[19] TcX AB, Detron HB and Monty Program KB. *MySQL v3.22 Reference Manual*, September 1998.

[20] Transvirtual Technologies Inc. http://www.transvirtual.com. *Kaffe*, January 1999.