Speculative Prefetching of *Induction Pointers*

Artour Stoutchinin[1], José Nelson Amaral[2], Guang R. Gao[3], Jim Dehnert[4], Suneel Jain[5], and Alban Douillet[3]

[1] STMicroelectronics, Grenoble, France
[2] Department of Computing Science, University of Alberta, Edmonton, AB, T6G-2E8, Canada
amaral@cs.ualberta.ca
http://www.cs.ualberta.ca/ amaral
[3] Computer Architecture and Parallel System Laboratory, University of Delaware, Newark, DE, USA
{ggao, douillet}@capsl.udel.edu
http://www.capsl.udel.edu
[4] Transmeta Co., Santa Clara, CA, USA dehnert@transmeta.com
[5] Hewlett-Packard Co., Cupertino, CA, USA sjain@cup.hp.com

Abstract. We present an automatic approach to prefetch data for linked list data structures. The main idea is based on the observation that linked list elements are frequently allocated at constant distance from one another in the heap. When these lists are traversed, a regular pattern of memory accesses with constant stride emerges. This regularity in the memory footprint of linked lists enables the development of a prefetching framework where the address of the element accessed in one of the future iterations of the loop is dynamically predicted based on its previous regular behavior.

We automatically identify pointer-chasing recurrences in loops that access linked lists. This identification uses a surprisingly simple method that looks for *induction pointers* — pointers that are updated in each loop iteration by a load with a constant offset. We integrate induction pointer prefetching with loop scheduling. A key intuition incorporated in our framework is to insert prefetches only if there are processor resources and memory bandwidth available. In order to conserve memory bandwidth we estimate the number of potential cache misses in a loop iteration. Our estimation algorithm is based on an application of graph coloring on a memory access interference graph derived from the control flow graph. We implemented the prefetching framework in an industry-strength production compiler, and performed experiments on eleven benchmark programs with linked lists. We observed performance improvements between 15% and 35% in three of them. For the programs that do not benefit from pointer prefetching, our profitability analysis significantly reduces performance degradations.

1 Introduction

Modern computers feature a deep memory hierarchy. The data move from main memory to processor register files through a number of levels of caches. Prefetch

```
1  max = 0;
2  current = head;
3  while(current != NULL) {
4    if(current -> key > max)
5      max = current -> key;
6    current = current -> next;
7   }
```

```
1  max = 0;
2  current = head;
   tmp = current;
3  while(current != NULL) {
4    if(current -> key > max)
5      max = current -> key;
6    current = current -> next;
     stride = current - tmp;
     prefetch(current + stride*k);
     tmp = current;
7   }
```

Fig. 1. A *p*ointer-chasing loop that scans a linked list without and with prefetching.

mechanisms can bring data into the caches before it is referenced. These mechanisms rely on the spatial and temporal locality of data that naturally occurs in regular programs. However, for irregular, pointer-based applications, prefetching tends to be less effective. Some critical issues that make the prefetching of pointer-based data more difficult include:

1. Pointer-based applications display poor spatial and temporal locality. Unlike array elements, consecutive data elements of a pointer-linked data structure do not necessarily reside in adjacent memory locations. Also, references to many unrelated data locations often intervene between reuses of linked list elements.
2. In general, the address of a data element accessed in iteration i is not known until iteration $i-1$ is executed, and thus cannot be prefetched in advance. This address dependence is known as the *pointer-chasing* problem.
3. Most pointer-based data structures are allocated in heap memory. Because the heap is typically a large memory space, and these structures can be anywhere in that space, heap allocation makes it difficult for compilers to reliably predict cache misses or perform reuse analysis. Ad-hoc generation of prefetches may result in performance degradation due to increased memory traffic and cache pollution.

Prefetching in conjunction with software pipelining has been successfully used in optimization of numerical loops [10]. In this work we perform *induction pointer prefetching* to optimize pointer chasing loops. Our objective is to use induction pointer prefetching to improve the software pipelining initiation interval and to prevent the software pipeline from being disrupted because of primary cache misses. Usually, the scheduler builds the software pipeline optimistically assuming cache hit load latencies. If a load misses in the cache, the pipeline is disrupted and loses its efficiency.

The induction pointer prefetching identifies *induction pointers* in pointer-chasing loops and generates code to compute their memory access stride and to

prefetch data. A loop contains a recurrence if an operation in the loop has a direct or indirect dependence upon the same operation from some previous iteration. The existence of a recurrence manifests itself as a cycle in the dependence graph. A *pointer-chasing recurrence* is a recurrence that only involves loads with constant offset and copy operations. We call the addresses used by loads involved in the pointer-chasing recurrences *induction pointers*. For instance, in the example in Figure 1, `current` is an induction pointer. The term comes from the fact that such loads are similar to induction variables in that the address for the load in the next iteration is provided by the execution of the load in the current iteration.

Our prefetch method is motivated by the observation that, although the elements of a linked list are not necessarily stored contiguously in memory, there tends to be a regularity in the allocation of memory for linked lists. If the compiler is able to detect a linked list scanning loop, it can assume that the accesses to the list elements generate a regular memory reference pattern. In this case, the address of an element accessed in iteration $i + k$ is likely to be the effective address of the element accessed in iteration i plus $k * S$, where S is the constant address stride. The compiler can then insert code that dynamically computes the stride S for induction pointers. Figure 1 shows a pointer chasing loop that scans a linked list before and after prefetch insertion. We use statements in the C language in Figure 1 for easy understanding. See [17] for the actual intermediate code used for prefetching.

Once the memory access stride of induction pointers is identified, data whose addresses are computed relative to the induction pointers are speculatively prefetched. The prefetch instructions issued do not cause exceptions and even if their address is miss predicted the program correctness is preserved. We perform a *profitability analysis* to establish whether prefetching is beneficial. Prefetching is only performed if the analysis indicates that there are unused processor resources and memory bandwidth for it. Our profitability analysis compares the lower bounds on the initiation interval imposed by data dependency constraints and by processor resource constraints. To prevent processor stalls due to prefetching, we also estimate the number of outstanding cache misses in any iteration of the loop. We use this estimate to avoid prefetching when the resulting number of outstanding cache misses would exceed the maximum number supported by the processor.

The research problem addressed in this paper can be posed as the set of open questions listed below. Although the problem of prefetching induction pointers, with and without hardware support, has been studied before (see Section 5), to the best of our knowledge the framework presented in this paper is the first that relies exclusively on compiler technology and addresses all the questions bellow.

- How to identify pointer-chasing recurrences using a low complexity algorithm? (see Section 2.1)
- How to decide when there are enough processor resources and available memory bandwidth to profitably prefetch an induction pointer? (see Section 2.5)

- How to efficiently integrate induction pointer prefetching with loop scheduling based on the above profitability analysis? This integration becomes complex for pointer chasing loops with arbitrary control flow. (see Section 3)
- How to formulate, algorithmically, the problem of maximizing the use of cache-memory bandwidth in non-blocking cache systems that support multiple outstanding cache misses? (see Section 2.4)
- How well does speculative prefetching of induction pointers work on an industry-strength state-of-the-art optimizing compiler? (see Section 4)

We describe our profitability analysis in section 2 and the prefetch algorithm in section 3. We present the wall-clock execution time measurements and the variation in the number of cache misses and TLB misses incurred with and without prefetching in section 4. Finally we discuss related research in section 5.

2 Induction Pointer Identification and Profitability Analysis

Issuing a prefetch in our framework requires up to three extra address computation instructions: a copy instruction to save the address referenced in the current iteration, a multiply (or add) instruction to compute the prefetch address, and the prefetch instruction itself. The insertion of these instructions in a program can have the following consequences:

1. There are more instructions to issue, consuming additional processor resources (computation units, issue slots, register file ports, etc).
2. There is more memory traffic to fetch the instructions, and potentially additional data traffic due to prefetch address miss-prediction.
3. The longer code might cause additional instruction cache misses.
4. Miss-predicted prefetching might displace useful data from the cache.

Our profitability analysis ameliorates consequences 1 and 2 above. In our experience, pointer chasing loops are short, therefore consequence 3 is not an important concern. In our current implementation, the compiler does not take into account consequence 4 when deciding whether to prefetch or not. Nonetheless the effects of cache pollution are reflected in the results presented in Section 4.

Ideally, we would like induction pointer prefetching to never cause any performance degradation. Therefore we should only issue a prefetch when there are enough under-utilized processor resources and memory bandwidth to ensure that the execution of the program will not slow down because of the insertion of prefetch. The execution time of a loop is determined by the rate at which new iterations of the loop can start execution. Therefore, we attempt to avoid performance degradation by estimating the constraints imposed by processor resources and by recurrent data dependencies on the initiation interval of each loop. Based on these constraints we decide when we should prefetch for a given induction pointer.

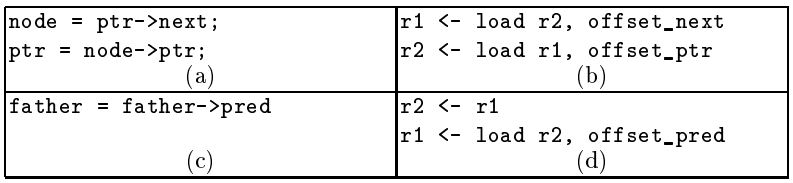

Fig. 2. Examples of code occurring in pointer-chasing loops.

The *initiation interval* (II) of a loop that contains a single control path is defined as the number of clock cycles that separate the issuing of the first instruction of two consecutive iterations of the loop [13]. The minimum initiation interval (MII) of a loop defines a lower bound on the initiation interval of instructions in the loop. In loops with multiple control paths, the interval between consecutive executions of an operation L depends on the control path executed in each iteration, and may change from iteration to iteration. Thus, for loops with multiple control paths, we can define the MII *of an operation* L, $MII(L)$ as the lower bound on the initiation of L imposed by all the recurrences that execute L.

If a load operation L is part of an induction pointer recurrence, we base our decision to prefetch for L on three estimates: (1) a conservative bound on its initiation interval; (2) an estimate of the increase in cache misses that may occur when we prefetch for L; and (3) an estimate of the resource usage in the control paths that execute L. Prefetching of L is deemed profitable only if it does not increase the initiation interval of L and if a potential extra cache miss caused by the prefetch will not result in a stall of the memory queue.

2.1 Identification of List Traversing Circuits

We identify circuits in the data dependence graph of a loop that are indicative of a linked list being traversed. Consider the code statements shown on Figure 2 (a) and (c) that are often used to traverse linked lists. These statements are translated by the SGI compiler into the corresponding assembly sequences shown on Figure 2 (b) and (d). Our key observation is that a circuit formed exclusively by loads (with some offset) and copy instructions is likely to be a pointer-chasing circuit.

We used the two patterns shown in Figure 2 for identification of the pointer-chasing circuits. Conceptually our algorithm finds all the elementary circuits in the loop's data dependence graph using Tarjan's algorithm [18] and then finds those circuits consisting exclusively of loads with constant offsets and copy instructions. Enumerating all circuits can be exponential in a general graph. Our implementation avoids enumeration by identifying only the pointer-chasing circuits. This has one important implication for the efficiency of our implementation: Tarjan's algorithm's complexity is $O(N * E * C)$, where N and E are respectively the number of nodes and the number of edges in the data depen-

dence graph, and C is the number of elementary circuits. Since in practice loops tend to have very few pointer-chasing circuits (one in most cases), our implementation practically achieves $O(N * E)$ complexity.

2.2 Recurrence Minimum Initiation Interval

The recurrence minimum initiation interval (recMII) of a loop is a lower bound imposed on the MII by recurrences (cyclic data dependences) among operations from multiple iterations of the loop. In classical software pipelining the recMII for a recurrence c is given by the quotient of the sum of the latencies of the operations in c, $latency(c)$ and the sum of the *iteration distances* of the operations in c, *iteration distance*(c) [13]. The iteration distance of each dependence in c is equal to the number of iterations separating the dependent operations.

$$recMII(c) = \left\lceil \frac{latency(c)}{iteration\ distance(c)} \right\rceil$$

When a loop L has a single control path, the recMII for the loop is computed by taking the maximum of the $recMII(c)$ over all the elementary circuits in the data dependence graph of the loop.

In this framework we are also interested in prefetching in loops that have multiple control paths. Therefore we must extend the concept of recMII to suit our purposes. To this end we define the recMII associated with each instruction L in the loop, $recMII(L)$. Instead of all recurrences in the loop, the $recMII(L)$ considers only the recurrences in which L participates. Therefore $recMII(L)$ is the maximum of $recMII(c)$ among all recurrences that execute L, and defines a *conservative bound* on the initiation interval of instruction L.

$$recMII(L) = \max_{c|L \in c} \{recMII(c)\}$$

When computing $latency(c)$ we assume that the loads in a pointer chasing recurrence incur cache misses. As prefetches are added, the prefetched loads are upgraded to cache hits, and the value of $latency(c)$ is re-computed. Therefore prefetching a load L reduces the $recMII(c)$ for all the recurrences that contain the load L.

2.3 Resource Minimum Initiation Interval

The resource usage of the operations along a control path of a loop imposes another lower bound on the rate at which operations in that path can be initiated, the resource MII. The usage of each resource in a control path is calculated by adding the resource usage of all operations in that path. A conservative bound on the initiation interval of operations in a basic block B is the maximum over lower bounds imposed by resource usage in each control path that executes B:

$$resMII(B) = \max_{p|B \in p} resMII(p)\}$$

where $resMII(p)$ is the lower bound on instruction initiation rate in the control path p.

The $resMII(B)$ is computed using the DAG longest path algorithm [3] on the control flow graph (CFG).[1] Each vertex in the CFG represents a basic block B. We associate a vector of weights, $w(B)$ to each basic block. Each element of $w(B)$ represents the usage of a processor resource by the basic block B. The longest path for a given processor resource r is the one that uses r the most. The $resMII(B)$ is given by the maximum longest path over all processor resources.

2.4 Cache/Memory Available Bandwidth

We define the *available memory bandwidth* of a basic block B, $M(B)$, as the number of additional memory references that can be issued in B without increasing the initiation interval of operations in B. As with processor resources, we use a conservative estimate of available memory bandwidth. $M(B)$ is defined as the minimum available bandwidth over all control paths that execute B:

$$M(B) = \min_{p:B \in \mathrm{p}} \{M(p)\}$$

To compute the available memory bandwidth of a control path p, $M(p)$, we estimate the number of expected cache misses, $m(p)$, in each control path of the loop. The available memory bandwidth of p is a function of $m(p)$ and the number of outstanding misses that the processor can have before it stalls, k. In our experiments we defined the available memory bandwidth of a control path p as the difference between the two: $M(p) = k - m(p)$.

2.5 Prefetch Profitability Condition

Given a basic block B that contains a memory reference L that is part of a pointer-chasing recurrence, the decision to prefetch for L is based on the prefetch profitability condition below:

Condition 1 (prefetch profitability condition) *Prefetching for a load L in a basic block B is profitable if the following condition holds:*

$$\left(resMII^P(B) \leq recMII^P(L)\right) \wedge \left(M(B) > 0\right)$$

where $resMII^P(B)$ and $recMII^P(L)$ are the resource MII of the basic block B and the recurrence MII for the load L after the prefetch sequence for the load is inserted in the code, and $M(B)$ is the available memory bandwidth of the basic block B before the prefetch sequence is inserted in the code.

Condition 1 states that prefetching for a load L in a basic block B is profitable only if the initiation interval of the loop (with prefetching for L) is still limited by

[1] We do not consider the loop's back edge when applying the longest path algorithm to the CFG. As a consequence, our current implementation of the profitability analysis ignores the use of processor resources carried over iterations.

recurrent data dependencies *and* there is enough bandwidth available between the cache and the memory for the basic block B. This is a conservative approach because, even when there is no available bandwidth, prefetching the correct address could improve performance by starting memory accesses earlier. We adopted such a conservative criterion because we are speculating the address that might be accessed in future iterations of a loop.

A more precise analysis could match each recurrence with the control path that it executes to estimate the balance between constraints imposed by data dependencies and resource usage. However such matching is expensive. Instead, we optimistically assume that the recurrence that determines $recMII(B)$ executes the most resource constrained control path to which B belongs.

2.6 Prefetching Field Loads

Prefetching for an induction pointer load enables prefetching of field loads that access memory through the same induction pointer. Thus, prefetching for a field load does not require additional address computations. Although prefetching of field pointers does not affect recurrences, it may still be beneficial to prefetch data for field loads because such prefetches will allow for the overlapping between memory accesses to fields and other computations performed by the loop. Prefetching of field loads is considered profitable if there are enough unused processor and memory resources for such prefetching in every control flow path affected. A field load prefetch is issued only if the field is not expected to lay in the same cache line as its induction pointer (see Section 3.1).

3 Prefetch Algorithm

The goal of the prefetch algorithm is to introduce prefetching whenever the extra usage of resources does not increase the MII of the loop beyond the constraint imposed by the recurrence MII. The algorithm proceeds as follows.

Prioritize the loads in pointer-chasing recurrences (step 2-4 in Figure 3) according to the frequency of execution of the basic blocks to which they belong. The "other loads" in step 3 are loads that are not part of a recurrence, are not loop invariant, and are not relative to a global pointer or to the stack pointer. These other loads are also candidates for prefetching and thus are added to the N list in step 3. The N list then becomes the end of the priority list O in step 4 of the algorithm. The loads in N are likely to be accesses to fields of the structures that formed the pointer chasing recurrence. Prefetch for the loads in N if there are enough resources available after the prefetch for the loads in pointer chasing recurrences. In step 5 mark the loads in the list O as cache misses. Observe that once a pointer-chasing recurrence is identified, it is reasonable to predict that without prefetching the loads of such pointers will result in cold cache misses.[2]

[2] Some exceptions to this rule include inner loops traversing short linked lists, or circular pointer chasing structures. Our framework does not attempt to identify such situations.

```
DoPrefetch(P, E, V)
1.  C ← pointer-chasing recurrences;
2.  R ← Prioritized list of induction pointer loads in C;
3.  N ← Prioritized list of other loads (not in C);
4.  O ← R + N
5.  Mark each o in O as a cache miss;
6.  for each L in O, L ∈ B
7.      do if recMII^P(L) ≥ resMII^P(B) and M(B) > 0
8.          then add prefetch for L to B;
9.               mark L as cache hit;
10.     endif
11. endfor
```

Fig. 3. Prefetch algorithm.

For each load operation in the priority list, $L \in B$, compute the available memory bandwidth of B, $M(B)$. If $M(B)$ is greater than zero, it means that a new cache miss caused by the prefetch instruction will not cause the processor to stall. Therefore there is unused memory bandwidth available for a prefetch instruction. Then insert the prefetch sequence in B and compute the recurrence MII of the load L, $recMII^P(L)$, as the maximum of the recMII of all recurrences to which L belongs, and the resource MII of B, $resMII^P(B)$, as the maximum MII for all processor resources over all the control paths to which B belongs. If $recMII^P(L) \geq resMII^P(B)$, whenever the loop executes B, it is still constrained by the recurrences and not by resource usage. In this case keep the prefetch sequence in B and mark L as a cache hit for future memory bandwidth availability estimates, otherwise remove the prefetch sequence.

The use of the most constrained recurrence and the most constrained control path to estimate $recMII^P(L)$ and $resMII^P(B)$ is an approximation used to avoid the high complexity of matching recurrences to control paths in the algorithm. Observe that inserting a prefetch sequence for a load L in a basic block B may change the resource use and the memory bandwidth availability for all basic blocks that share a control path with B. It also changes the $recMII$ for the recurrences to which L belongs. Therefore these quantities must be computed for each operation that we consider for prefetching.

3.1 Computation of the Memory Bandwidth Availability

The available memory bandwidth of a basic block B of a loop, $M(B)$, is a function of the maximum number of cache misses that may occur on any of the control paths that execute B. Observe that the available memory bandwidth of all operations that lie on the same control path changes when a prefetch sequence is inserted in the path. We estimate the number of cache misses that can occur in one iteration of the loop based on the coloring of the cache miss interference graph.

Definition 1. *The* **miss interference graph** *is an undirected graph $G(V, E)$ formed by a set of vertices, V, representing the memory operations in the loop, and a set of edges, E, representing the* **interference** *relationship between the memory operations. Two memory operations interfere if they may both cause a cache miss in the same loop iteration.*

When the cache miss interference graph is colored, memory operations that interfere with each other are assigned distinct colors. Thus, the minimum number of colors necessary to color the miss interference graph corresponds to the maximum number of misses that are expected to occur in one iteration of the loop, regardless of the control path executed by that iteration.

We use the following set of heuristic interference rules to build the miss interference graph:

- memory references that are relative to global scope addresses do not interfere with any other memory references (we assume that they hit in the cache);
- memory references that are relative to the stack pointer do not interfere with any other memory references (we assume that they hit in the cache);
- memory references where the memory address is loop invariant do not interfere with any other memory references;
- two memory references of the form **load/store base + constant offset** do not interfere if the bases are the same and the difference in their offsets is less than K, where K is a parameter in our algorithm (we used K = 32, i.e. the cache line size).
- two memory references that are executed in mutually exclusive control paths do not interfere with each other.

The available memory bandwidth for a basic block B, is $M(B) = k - N$, where N is the number of distinct colors necessary to color the set of loads in B and all its adjacent nodes in the cache miss interference graph, and k is the number of outstanding cache misses that the processor can support before it stalls.

4 Performance Evaluation

We implemented our prefetch framework in the SGI MIPSpro Compiler suite. This suite consists of highly-optimizing compilers for Fortran, C, and C++ on MIPS processors. It implements a broad range of optimizations, including inter-procedural analysis and optimization, loop nest rearrangement and parallelization, SSA-based global optimization, software pipelining, global and local scheduling, and global register allocation [2, 8, 15].

We compiled a set of benchmarks, and executed the resulting code on an SGI Onyx system, based on the MIPS R10000 microprocessor. We measured wall-clock time for each benchmark under the IRIX 6.5 operating system with the machine running in a single user mode. The results reported are an average of three runs, and there was no noticeable difference on the measurements obtained in each of the three separate runs of the benchmarks. Our experimental results, in the follow subsections, evidence that

- The prefetch technique presented in this paper achieves significant speedup for programs that spend significant time on pointer chasing loops (see Table 1).
- The profitability analysis is effective for improving the gains of prefetching by balancing the reduction in number of cycles in a loop critical path and the increased processor resources and memory bandwidth requirements due to prefetching (see Tables 1 and 2).
- There are programs with pointer chasing loops for which prefetching is not effective. We measured the effect of prefetching on cache misses and TLB misses on this program and present an analysis of these programs behavior [17]. For most of these programs, the profitability analysis prevents prefetching from resulting in performance degradation (see Table 2).

4.1 Experimental Framework

The Silicon Graphics Onyx machine that we used for the experiments has a 195 MHz MIPS R10000 processor, 32 KB, two-way associative, non-blocking, primary cache and 1MB secondary cache. Implementing our prefetch technique for an out-of-order processor such as the MIPS R10000 processor has advantages and disadvantages. The out-of-order scheduler of the MIPS R10000 dynamically pipelines loops that cannot be pipelined by existing compilers. The pipelining of these loops is essential in order for the improved MII with prefetching to be reflected in the performance of the code. On the other hand, the out-of-order issuing of instructions makes the measurement of performance difficult for two reasons: first, the dynamic scheduler may move prefetches closer to their target loads reducing prefetch efficiency; second, an out-of-order processor can reorder memory operations, and therefore affect the cache miss behavior.

Test Programs To test our prefetch technique, we used two SPEC CINT95 benchmarks (126.gcc and 130.li), seven SPEC CINT2000 (181.mcf, 197.parser, 300.twolf, 175.vpr, 252.eon, 253.perlbmk, and 254.gap) and two applications (mlp is a perceptron simulator and ft is a minimum spanning tree algorithm). For the benchmarks in the SPEC benchmark suite we used the reference input. We chose these programs because they have loops that traverse linked lists. Because our technique is designed to improve such loops, applying it to benchmarks that do not contain linked lists would be pointless.

4.2 When Pointer Prefetching Works

Table 1 shows five programs that benefit from the prefetching of induction pointers, the table shows the execution time of each program using three prefetch strategies: (1) without prefetching, (2) prefetching without profitability analysis, and (3) induction pointer prefetching with the profitability analysis.[3] In the

[3] At this point we do not have the results for the SPEC CINT2000 benchmarks without profitability analysis because the machine was not available for an extra run, we will include such numbers in the camera ready version of the paper.

Benchmark	Execution Time			Performance Improvement	
	No Prefetch	No Prof. Analysis	With Prof. Analysis	No Prof. Analysis	With Prof. Analysis
181.mcf	3,396 sec.	2,854 sec.	2,699 sec.	16.0 %	20.5 %
ft	517 sec.	436 sec.	333 sec.	15.6 %	35.5 %
mlp	632 sec.	580 sec.	538 sec.	8.3 %	14.9 %
175.vpr	1,771 sec.	-	1,761 sec.	-	0.6 %
130.twolf	2,540 sec.	2,657 sec.	2,531 sec.	-4.6 %	0.4 %

Table 1. Execution time in seconds; performance improvement for programs that benefit from prefetching.

measurements without profitability analysis we prefetched whenever we found a pointer chasing loop without verifying whether the additional demand on memory bandwidth and processor resources would result in an increased MII. When the profitability analysis is employed, the gains of prefetching are significantly higher, with improvements of more than 15% (with a remarkable 35% for ft) on top of all the standard optimizations of an industry-strength compiler. The profitability analysis transforms a performance degradation of 4.6% into a performance improvement of 0.4% for 130.twolf.

The execution times shown in Table 1 demonstrate that there is a significant improvement in the performance of mcf, ft, and mlp. In mcf a single loop accounts for 40% of the program execution time. This loop has two induction pointers and both of them access memory locations at regular intervals that exceed the cache line size. The iteration count of this loop is comparatively large, up to 400 iterations, and the control path taken most often is constrained by the pointer chasing recurrence. Prefetching is very effective in hiding this loop's primary cache miss latencies and has a significant impact on its performance. *ft* spends about 80% of its time in a loop that traverses a linked list of vertices of a graph. No other computation is done in that loop and the loop's initiation interval is constrained only by the pointer-chasing recurrence. The *mlp* program has two linked list loops where the majority of the execution time is spent. One of these loops updates the weights of the synapses of neurons in a multi-layer perceptron, while the other updates the gradient used to back-propagate the output error. The initiation interval of these loops is constrained by the pointer chasing recurrences. The trip count is moderate, 25 iterations on average, but large enough that prefetching makes a significant difference.

4.3 When Pointer Prefetching Does Not Help

Table 2 displays the results for six programs that have pointer chasing loops but do not benefit from speculative induction pointer prefetching. For these programs we are interested on minimizing any performance degradation caused by prefetching. As the table shows the profitability analysis is effective on the reduction of this degradation for gcc and parser.

Benchmark	Execution Time			Performance Improvement	
	No Prefetch	No Prof. Analysis	With Prof. Analysis	No Prof. Analysis	With Prof. Analysis
254.gap	1,174 sec.	-	1,207 sec.	-	-2.8 %
130.li	285 sec.	293 sec.	292 sec.	-2.8 %	-2.5 %
253.perlbmk	2,062 sec.	-	2,104 sec.	-	-2.0 %
252.eon	949 sec.	-	959 sec.	-	-1.1 %
197.parser	2,180 sec.	2,245 sec.	2,189 sec.	-3.0 %	-0.5 %
126.gcc	122 sec.	123 sec.	122 sec.	-0.7 %	-0.1 %

Table 2. Execution time in seconds; performance degradation for programs that do not benefit from prefetching.

Although *gcc* uses linked lists extensively, the impact of prefetching is not visible because these linked lists are short and rarely suffer primary cache misses. In gcc, prefetching disturbs the virtual memory mechanism and causes a significant increase in the number of TLB misses. li spends much of the time in a number of loops that operate on linked lists, but the input that we used (SPEC95) only executes 1-2 iterations at a time, which might be not enough to benefit from our method. The 3% performance degradation in li, and the 20% increase in the number of cache misses when prefetch is inserted indicates that li might not have stride regularity, and that miss-predicted addresses for prefetching are displacing useful data from the cache. *parser* has a number of linked-list loops. However, these lists are either hash table lists, i.e. random in memory, or their location in memory is randomized by the way the memory allocation in this program is done. Thus prefetching is ineffective and results in moderately higher L1 cache misses and TLB misses. Nonetheless, with profitability analysis the performance degradation for parser is only 0.5%.

5 Related Work

Compiler prefetching for array-based numeric applications takes advantage of the data locality and of the compiler ability to analyze numerical loops [10]. Application of prefetching to dynamically allocated irregular data structures is more difficult. Such data structures can not be efficiently analyzed by the compiler with respect to locality. Therefore in order for prefetching to be effective it is often necessary to speculatively predict addresses to drive prefetching. Many prefetching techniques have been proposed that try to address these problems.

Mowry and Luk use profiling to identify potential cache misses [11]. Ozawa *et al.* develop a compiler analysis combined with instruction scheduling based on the observation that certain kinds of memory references that represent a small fraction of static instructions tend to be responsible for the majority of cache misses [12]. Lipasti *et al.* use the fact that often the data at an address passed as a parameter to a function call suffers a cache miss to issue prefetch for such cases [6].

Address speculation has been typically used in hardware prefetching schemes. As in this work, they exploit the fact that often the addresses referenced by loads and stores follow an arithmetic progression. By keeping track of the last effective address and of the address stride, previously unseen addresses are speculatively predicted [1, 4, 5]. In particular, Selvidge [16] and Mehrotra [9] noticed the regularity in memory streams generated by linked list accesses. A similar hardware approach is to reproduce the address generation process in hardware and to perform it in advance of other computations [14].

Our approach is different from the one proposed by Luk and Mowry both in the identification of pointer chasing loops and in the implementation of prefetching [7]. They use high level declarations to identify Recursive Data Structures (RDS) records and points-to analysis to detect when a pointer to a record is assigned a value that was obtained from the dereference of a pointer to the same record. The implementation of such induction pointer identification requires high-level information from compiler front-end, and expensive pointer analysis for identifying a recursive data structure traversal. Their approach does not allow the application of prefetching exclusively to linked list traversals. In contrast our technique to identify induction pointers detects recurrence cycles associated with pointer-chasing. The complexity of our induction pointer detection is $O(n^2)$ in the number of nodes in the DDG of a loop, and thus is not nearly as complex as a points-to analysis. Notice that for the case of regular strides, Luk and Mowry's data linearization scheme requires that the data be re-mapped to contiguous memory location in order to use pointer arithmetic to compute the addresses. In our framework no data re-mapping is required.

6 Conclusion

Prefetching for pointer-based applications has been recognized as a difficult problem because the compiler does not know what to prefetch and when to do so. In this paper we present a compiler prefetch method that is applicable to linked lists. We identify candidates for prefetching as data accessed through induction pointers and propose a profitability analysis which effectively determines when such prefetching will be beneficial. The prefetch technique presented in this paper is remarkably simple and quite effective for some benchmarks, resulting in gains of performance of 15-35% over their performance after standard compiler optimizations, while displaying minimal performance degradation for others. This simplicity — a small set of extra instruction for dynamic address calculation, no data re-mapping, and no hardware support — allowed for an effective implementation in an industry-strength compiler. Moreover, there was no noticeable change in the time required to compile the benchmarks that we tested when the prefetch analysis was executed. Thus using our framework, a compiler optimization flag can be implemented in existing compilers. Such a flag would allow existing processor architectures to deliver significant speedup for programs that access regular strided data-structures.

References

1. T. F. Chen and J. L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44:609 – 623, May 1995.
2. F. Chow, S. Chan, R. Kennedy, S-M Liu, R. Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *International Conference on Programming Languages Design and Implementation*, pages 273 – 286, 1997.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press; McGraw-Hill Book Company, Cambridge, Massachusetts; New York, New York, 1990.
4. J. Fu and J. Patel. Stride directed prefetching in scalar processors. In *International Symposium on Microarchitecture*, pages 102 – 110, 1992.
5. J. Gonzales and A. Gonzales. Speculative execution via address prediction and data prefetching. In *International Conference on Supercomputing*, pages 196 – 203, 1997.
6. M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *International Symposium on Microarchitecture*, pages 231 – 236, 1995.
7. C. K. Luk and T. Mowry. Compiler based prefetching for recursive data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222 – 233, 1996.
8. S. Mantripragada, S. Jain, and J. Dehnert. A new framework for integrated global local scheduling. In *Conference on Parallel Architectures and Compilation Techniques*, pages 167–174, Paris, France, October 1998.
9. S. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
10. T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.
11. T. Mowry and C. K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *International Symposium on Microarchitecture*, pages 314 – 320, 1997.
12. T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *International Symposium on Microarchitecture*, pages 243 – 248, 1995.
13. B. Rau. Iterative modulo scheduling. Technical Report HPL-94-115, HP Laboratories, 1995.
14. A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115 – 126, 1998.
15. J. Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *International Conference on Programming Languages Design and Implementation*, pages 1–11, Philadelphia, PA, May 1996.
16. C. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, MIT, 1992.
17. A. Stoutchinin, J. N. Amaral, G. R. Gao, J. Dehnert, and S. Jain. Automatic prefetching of induction pointers for software pipelining. Technical Report 37, November 1999.
18. R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211 – 216, September 1973.