Speculative Prefetching of *Induction Pointers*

Artour Stoutchinin[1], José Nelson Amaral[2], Guang R. Gao[3], James C. 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
[6] Stoutchinin, Dehnert, and Jain were at SGI when most of this research was conducted.

Abstract. We present an automatic approach for prefetching 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 linked 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 estimate available memory bandwidth we calculate the number of potential cache misses in one 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 ten benchmark programs with linked lists. We observed performance improvements between 15% and 35% in three of them.

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

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 289–303, 2001.

```
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 is often 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 [12]. 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. See [19] 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 mispredicted 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 upper bounds on the initiation rate 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 avoid prefetching when this number exceeds the maximum number supported by the processor.

This paper addresses 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:

- 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? Such 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

Ideally, we would like speculative prefetching to never cause any performance degradation. Induction pointer prefetching may lead to performance degradation as a result of one of the following: increased processor resource requirements (computation units, issue slots, register file ports, etc); increased memory traffic due to potential fetching of useless data; increased instruction cache misses; and potential displacement of useful data from the cache.

Our profitability analysis addresses the problems of not degrading performance due to the increase in required processor resources and in the memory traffic. In our experience, pointer chasing loops are short, therefore instruction cache misses are not an important concern. In our current implementation, the profitability analysis does not take into account the cache pollution problem. Nonetheless the effects of cache pollution are reflected in the results presented in Section 4.

The execution time of a loop is determined by the rate at which iterations can be initiated. We attempt to avoid performance degradation by estimating the constraints imposed by processor resources and by recurrent data dependencies on the initiation rate of each loop. Based on these constraints we decide when we should prefetch for a given induction pointer.

The *initiation interval* (II) of a loop with a single control path is defined as the number of clock cycles that separate initiations of consecutive iterations of the loop. The minimum initiation interval (MII) defines an upper bound on the initiation rate of instructions in a loop. The MII is computed as the maximum of *recurrence MII* and the *resource MII* [5].

In loops with multiple control paths, the interval between consecutive executions of any given operation depends on the control path executed in each iteration, and may change from iteration to iteration. A conservative upper bound on the initiation rate of an operation L corresponds to the maximal MII of all control paths that execute L.

We base our decision to prefetch data for a load instruction that belongs to an induction pointer recurrence on three estimates: (1) an estimate of the conservative upper bound on its initiation rate; (2) an estimate of the potential

`node = ptr->next;`
`ptr = node->ptr;`
(a)
	`r1 <- load r2, offset_next`
	`r2 <- load r1, offset_ptr`
	(b)

`father = father->pred`
(c)
	`r2 <- r1`
	`r1 <- load r2, offset_pred`
	(d)

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

increase in cache misses; and (3) an estimate of the potential resource usage increase. Prefetching is deemed profitable if (i) it does not decrease the conservative upper bound on load's initiation rate, and (ii) a potential extra cache miss caused by the prefetch does not result in a stall of the cache fetching mechanism.

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 [20] 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 dependence 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.

The initiation rate of operations in list traversing circuits is bounded by induction pointer recurrences and the resource usage. A precise determination of the such bound should match each control path with recurrences that it executes. However, such matching is expensive. Instead, we consider that the initiation rate of an operation is limited by the most resource constrained control path in which this operation executes, and by the most constraining recurrence to which this operation belongs.

2.2 Recurrence Bound on the Initiation Rate

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) [5,15]. 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 rate 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 optimistically upgraded to cache hits, and the value of $latency(c)$ is recomputed. Therefore prefetching a load L reduces the $recMII(c)$ for all the recurrences that contain the load L.

2.3 Resource Bound on the Initiation Rate

The resource usage of the operations along a control path of a loop imposes another upper bound on the rate at which operations in that path can be initiated, the resource minimal initiation interval (resMII). 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 rate of operation L in basic block B is the maximum over lower bounds imposed by resource usage in each control path that executes B:

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

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

The $resMII(B)$ for each basic block is computed using the DAG longest path algorithm [4] 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 decreasing the initiation rate of operations in the loop. 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 potential cache misses, $m(p)$, in each path p 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) *Prefetching for a load L in a basic block B is considered profitable if the following condition holds:*

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

where $resMII^P(L)$ and $recMII^P(L)$ are the resource and recurrence conservative lower bounds on initiation interval for the load L after the prefetch sequence 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 rate of L is still limited by recurrences in the data dependence graph *and* the potential cache miss introduced by the prefetch will

[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.

```
DoPrefetch(P, E, V)
1.  C ← pointer-chasing recurrences;
2.  R ← Prioritized list of induction pointer loads in C;
3.  N ← Prioritized list of field 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 S(B) > 0
8.          then add prefetch for L to B;
9.               mark L as cache hit;
10.     endif
11. endfor
```

Fig. 3. Prefetch algorithm.

not block memory fetching from the cache. This is a conservative approach because, even when there is no available bandwidth between the cache and the main memory, prefetching at the correct address could improve performance by starting memory accesses earlier.

2.6 Prefetching Field Loads

Prefetching for an induction pointer load enables prefetching of [*field* loads. Field loads access memory through induction pointers but are not part of the recurrence cycles. Prefetching for the load of an structure field does not require additional address computations. Although prefetching of fields 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 affected control flow path. A field load prefetch is issued only if the field is not expected to lie 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 reduce the initiation rate of operations in the loop beyond the constraint imposed by loop recurrences.

We 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. In step 5 we 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]

For each load operation L in the priority list, we compute the available memory bandwidth of basic block B to which L belongs, $M(B)$. 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. In this case we insert the prefetch operations in B and compute $recMII^P(L)$ and $resMII^P(L)$. If $recMII^P(L) \geq resMII^P(L)$, whenever the loop executes L, its initiation rate is still constrained by the recurrences and not by resource usage. In this case we mark L as a cache hit for future memory bandwidth availability estimates, otherwise we remove prefetch operations from B.

Inserting prefetch operations for a load L in a basic block B may change the resource usage and the memory bandwidth availability for all basic blocks that share a control path with B. It also changes the $recMII$ of recurrences to which L belongs. Therefore these quantities must be recomputed 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 may 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 a global pointer do not interfere with any other memory references (we assume that they hit in the cache);

[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.

- 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 at address **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

In order to evaluate our prefetch technique, we have collected 10 programs that spend a significant portion of its execution time executing loops that traverse linked lists. Our test suite includes two SPEC CINT95 benchmarks (126.gcc and 130.li), seven SPEC CINT2000 (181.mcf, 197.parser, 300.twolf, 175.vpr, 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. Table 1 shows the execution time of each program using three prefetch strategies: no prefetching, prefetching without profitability analysis (all induction pointers in all pointer chasing loops are prefetched), and induction pointer prefetching with the profitability analysis.

Preliminary evaluation results show that (1) identification of the induction pointers is a good basis for prefetching; (2) the prefetching technique presented in this paper achieves good speedups for programs that spend significant time in pointer chasing loops; and that (3) balancing the loop recurrences and the processor resource usage is necessary in order for prefetching to be effective.

4.1 Experimental Framework

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 [3,10,17].

We performed our experiments on an SGI Onyx machine with a 195 MHz MIPS R10000 processor, 32 KB, two-way associative, non-blocking, primary

Table 1. Execution time in seconds with and without prefetching.

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.	1,765 sec.	1,761 sec.	0.3 %	0.6 %
130.twolf	2,540 sec.	2,657 sec.	2,531 sec.	-4.6 %	0.4 %
254.gap	1,174 sec.	1,226 sec.	1,207 sec.	-4.4 %	-2.8 %
130.li	285 sec.	293 sec.	292 sec.	-2.8 %	-2.5 %
253.perlbmk	2,062 sec.	2,131 sec.	2,104 sec.	-3.3 %	-2.0 %
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 %

cache and 1MB secondary cache. 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.

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. Pipelining these loops is essential in order for the improved iteration rate 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 accesses, and therefore affect the cache miss behavior.

Finally, the prefetch distance in our experiments has been set to prefetch 2 iteration in advance. We found that this distance is the most effective for improving the L1 cache performance. Prefetching 1 iteration in advance, on the other hand, reduces performance penalty when prefetching is counterproductive.

4.2 When Pointer Prefetching Works

The prefetching algorithm presented in this paper achieved significant (over 15%) improvement in performance of the mcf, ft, and mlp (with a remarkable 35% for ft) on top of all the standard optimizations of an industry-strength compiler.

We noticed that a very small number of loops are responsible for most of the performance gain in those programs. For example, in the 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 rate 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 rate 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

On the other hand, speculative induction pointer prefetching is not as effective in a number of programs. The two main reasons for this are: (1) short loop trip counts, and (2) irregular memory access pattern caused by the control flow in the loop. In such cases, our profitability analysis has been effective in keeping the negative impact of prefetching reasonably small.

For example, although the *gcc* extensively uses linked lists, prefetching does not have significant impact because those lists are short and rarely suffer primary cache misses. *li* spends much of its time in a number of pointer chasing loops that operate on trees. Tree traversal using loops rather than recursive function calls resembles the traversal of linked lists. Tree traversal does not have stride regularity though, and speculative prefetching is counterproductive (we measured a 2.5% performance degradation and 20% increase in the number of cache misses when prefetching is added). Another program, *parser*, has a number of linked-list loops. However, these lists are either hash table lists (randomly placed in memory), or their location in memory is randomized by the memory allocator. 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%.

These results suggest an improvement to our method of identification of memory references with potentially regular stride: complex control flow in a loop is a good indication that the stride may be irregular and prefetching must be avoided. This is subject for future work. On the other hand, use of profiling information would also be helpful in identifying loops with irregular strides or short trip counts.

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 [12].

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 information to identify potential cache misses [13]. 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 [14]. Finally, 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 [8].

Address speculation has typically been 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 [1,6,7]. In particular, Selvidge [18] and Mehrotra [11] 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 [16].

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 [9]. They use high level declarations to identify Recursive Data Structures (RDS) 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 the 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.

Chilimbi, Hill, and Larus propose the use of cache-conscious data structures to improve the caching of pointer-based data structures. They propose to improve locality by re-engineering the allocation of such structures [2].

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 in 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 remapping, 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, with our framework, a compiler optimization option can be implemented in existing compilers. Such a option would allow existing processor architectures to deliver significant speedup for programs that access data structures with regular stride.

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. T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, December 2000.
3. 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.
4. 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.
5. J. C. Dehnert and R. A. Towle. Compiling for the cydra 5. *The Journal of Supercomputing*, 7:181–227, May 1993.
6. J. Fu and J. Patel. Stride directed prefetching in scalar processors. In *International Symposium on Microarchitecture*, pages 102 – 110, 1992.
7. J. Gonzales and A. Gonzales. Speculative execution via address prediction and data prefetching. In *International Conference on Supercomputing*, pages 196 – 203, 1997.
8. 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.
9. 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.
10. 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.
11. S. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
12. T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

13. 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.
14. 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.
15. B. Rau. Iterative modulo scheduling. Technical Report HPL-94-115, HP Laboratories, 1995.
16. 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.
17. 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.
18. C. Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, MIT, 1992.
19. 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.
20. R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211 – 216, September 1973.