University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Automatic Prefetching of Induction Pointers for Software Pipelining

Artour Stoutchinin
José Nelson Amaral
Guang R. Gao
Jim Dehnert†
Suneel Jain†

CAPSL Technical Memo 37
November 12, 1999

Copyright © 1999 CAPSL at the University of Delaware

†Silicon Graphics Inc.

Abstract

We present an automatic approach to prefetch data for linked list data structures. The main idea is based on the observation that code operating on linked lists frequently generates a regular pattern of memory references. For example, in a loop that traverses a linked list, the effective addresses of the list elements accessed from one iteration to another may be x, $x + 40$, $x + 80$, ... This regularity in the memory footprint of linked lists inspired us to develop a prefetching approach where the address of the element accessed in one of the future iterations of the loop is speculatively predicted based on its previous regular behavior.

The main contributions of this paper include:

1. Automatic identification of pointer-chasing recurrences in loops that access linked lists. This identification can be achieved by a surprisingly simple method: simply looking for *induction pointers* — pointers that are updated in each loop iteration by a load with a constant offset.
2. Integration of induction pointer prefetching with loop scheduling. Key intuition in our framework is to insert prefetches only if there are extra processor resources and memory bandwidth available. In order to achieve that we estimate the number of potential cache misses in one loop iteration. Our algorithm is based on an application of graph coloring on a memory access interference graph derived from a control flow graph.
3. An implementation of the proposed scheme in an industry-strength production compiler. We have performed experiments on seven benchmark programs with linked lists and observed considerable performance improvements in three of them.

Contents

1 Introduction **1**

2 Motivation **2**

3 Overview of the Approach and the Running Example **4**

4 Profitability Analysis **5**

5 Prefetching Algorithm **8**
5.1 Identification of List Traversing Circuits 9
5.2 Address Update Sequence 9
5.3 Computation of Res_mii(B) 10
5.4 Computation of Mem_mii(B) 10

6 Performance Evaluation **11**
6.1 Experimental Framework 11
6.1.1 MIPSpro Compiler 12
6.1.2 Test Programs 13
6.2 Performance Measurements 14

7 Related Work **15**

8 Conclusion **16**

List of Figures

1 A *pointer-chasing* loop that scans a linked list 3
2 Code the creates the linked list 3
3 A portion of code taken from the **mcf** program 4
4 A portion of code taken from the **mcf** program 5
5 Prefetch code added to the **mcf** loop 6
6 Prefetching algorithm 8
7 Examples of code occurring in pointer-chasing loops 9
8 Prefetch sequence for the R10000 processor 10
9 Execution time of benchmarks without prefetching, with three prefetch strategies 12
10 Percentage improvement of each strategy 13

List of Tables

1 Test programs 13

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 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 less effective. Critical issues in prefetching of pointer-based data 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. This 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 propose using *induction pointer prefetching* in conjunction with software pipelining in optimization of pointer chasing loops. The objective of the induction pointer prefetching is 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*. 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 prefetching 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 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.

Once the memory access stride of induction pointers is identified, data whose addresses are computed relative to the induction pointers are speculatively prefetched. In order to prefetch data ahead of time prefetch instructions are issued. Prefetch instructions 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 there are unused processor resources and memory bandwidth for it. For the purpose of profitability analysis we distinguish induction pointer loads and field loads. The former update values of induction pointers and the latter do not. For induction pointer loads, in addition to the prefetch instruction our algorithm inserts the *address update sequence* that dynamically computes the memory stride and the predicted future address of the load. The address of a field load differs from corresponding induction pointer by a constant offset. Thus address update sequence generated for an induction pointer can be used for prefetching data for field loads.

We have implemented our methodology in the SGI MIPSpro compiler and tested it using a set of benchmark programs with linked lists. We observed wall-clock time improvements of up to 20% and very little performance degradation in some benchmarks. The improvements of 15 to 20% in the execution time is over the fully optimized code.

The rest of the paper is organized as follows. In Section 3 we present an overview of our prefetch methodology while discussing a motivating example. In Section 4 we describe our profitability analysis. In Section 5 we describe the prefetching algorithm. In Section 6 we present the results of our performance evaluation, and in Section 7 we discuss related work.

2 Motivation

We motivate a technique to find regularity in apparently irregular data access, by examining a simple piece of code. Most compilers might have difficulties implementing prefetching for the typical scanning of a linked list presented in Figure 1. The while loop in the figure scans the linked list to find the maximum value for the key. To prevent the load in line 6 from suffering a cache miss, a prefetch instruction can be added to move the necessary data to the L1 cache. In order to issue a prefetch, the compiler must know the address that will be accessed at iteration i at some iteration $i - k$ — the value of k is determined by the latency to fetch a value to the L1 cache and by the amount of computation computed in the loop. However this address is only known when the line 6 is executed in iteration $i - 1$. We call a loop such as the one in Figure 1 a *pointer-chasing loop*.

We observe that, although the elements of a linked list are not necessarily stored con-

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

Figure 1: A *pointer-chasing* loop that scans a linked list

tiguously in memory, there tends to be a regularity in the allocation of memory for linked lists. For instance, the while loop presented in Figure 2 is a typical creation for the linked list scanned in the loop of Figure 1. As long as the amount of memory allocated by the function `other_memory_allocations()` in line 11 is constant, the dynamic memory allocator will place the elements of the list at regular intervals in the memory. Therefore, if the compiler is able to detect a linked list scanning loop, such as the one presented in Figure 1, it can assume that the accesses to the list elements generate a regular memory reference pattern. In this case, the effective address of an element accessed at iteration $i + k$ is the effective address of the element accessed at iteration i plus $k * S$, where S is the constant address stride. The compiler can then insert code that dynamically computes the stride S and use it to prefetch data at the address that is likely to be referenced in some future iteration.

```
 8      curr = head = (item)malloc(sizeof(item));
 9      while((curr->key = get_key()) != NULL) {
10        curr -> next = curr = (item)malloc(sizeof(item));
11        other_memory_allocations();
12      }
13      curr -> next = NULL;
```

Figure 2: Code the creates the linked list

Although our study indicated that often the memory access stride in linked lists is equal the size of an element of the list, it is not always the case. For instance, if memory is allocated to unrelated data (as in the example of Figure 2), we might have a constant stride that is different from the element size. Moreover the same loop might have different strides when scanning different linked lists. Therefore, the compiler must generate code to calculate the stride dynamically. In our approach, a *prefetch sequence* is added to the loop. The prefetching sequence is specific to the target architecture, and is formed by instructions that calculate memory access stride and the prefetch instruction itself. Observe that our approach is robust to changes in the stride between calls to the same function that contains the pointer chasing loop, as well as to changes in the stride in a single loop trip, as long as they are not frequent.

3 Overview of the Approach and the Running Example

In this section we present an overview of our prefetching method and discuss its application to a pointer-chasing loop from the **mcf** program[1]. We profiled the **mcf** program on an SGI Onyx machine with 32 KB primary and 1MB secondary caches and observed that the program spends most of its time in the loop shown in Figure 3. This loop is also responsible for nearly half of the L1 cache misses in the program.

```
 1      while (arcin) {
 2        tail = arcin->tail;
 3        if (tail->time + arcin->org_cost > latest) {
 4          arcin = (arc_t *)tail->mark;
 5          continue;
          }
 6        arc_cost = tail->potential + head_potential;
 7        if (red_cost < 0) {
 8          if (new_arcs < MAX_NEW_ARCS) {
 9            insert_new_arc (arcnew, new_arcs, tail, head,
                                      arc_cost, red_cost);
10            new_arcs++;
            }
11          else if ((cost_t)arcnew[0].flow > red_cost)
12            replace_weaker_arc( arcnew, tail, head,
                                    arc_cost, red_cost );
          }
13        arcin = (arc_t *)tail->mark;
        }
```

Figure 3: A portion of code taken from the **mcf** program

The loop in Figure 3 iteratively visits arcs of a graph. Data in the `arcin->tail` field (line 2) contains the address of a graph vertex. The vertex data structure contains the address of an arc to be visited in the next iteration, the `tail->mark` field (line 4 and 13). We observed that the memory accesses for `arcin` and `tail` had a constant stride of -96 and -60 bytes respectively. That is, the address of `arcin` in iteration $i+1$ is equal to the address of `arcin` in iteration i minus 96.

Figure 4 shows the intermediate compiler representation and the control flow graph (CFG) of the mcf inner loop. The expressions `arcin->tail` and `tail->mark` become (`t228 = lw 0x0(t226)`) in B1, and (`t226 = lw 0x34(t228)`) in B2 and B7, respectively, where t228 and t226 are temporary names used before register allocation. There are two pointer-chasing circuits: the load in line 1 depends on the load in line 7 that depends on the load in line 1, and the load in line 1 depends on the load in line 15 that depends the on load in line 1. `t226` and `t228` are, thus, induction pointers and the two loads above are induction pointer loads. The rest of the loads whose address is calculated relative to `t226` and `t228`, for example load `t229`

[1]Minimal cost flow optimizer from Konrad-Zuse Informatics Center in Berlin.

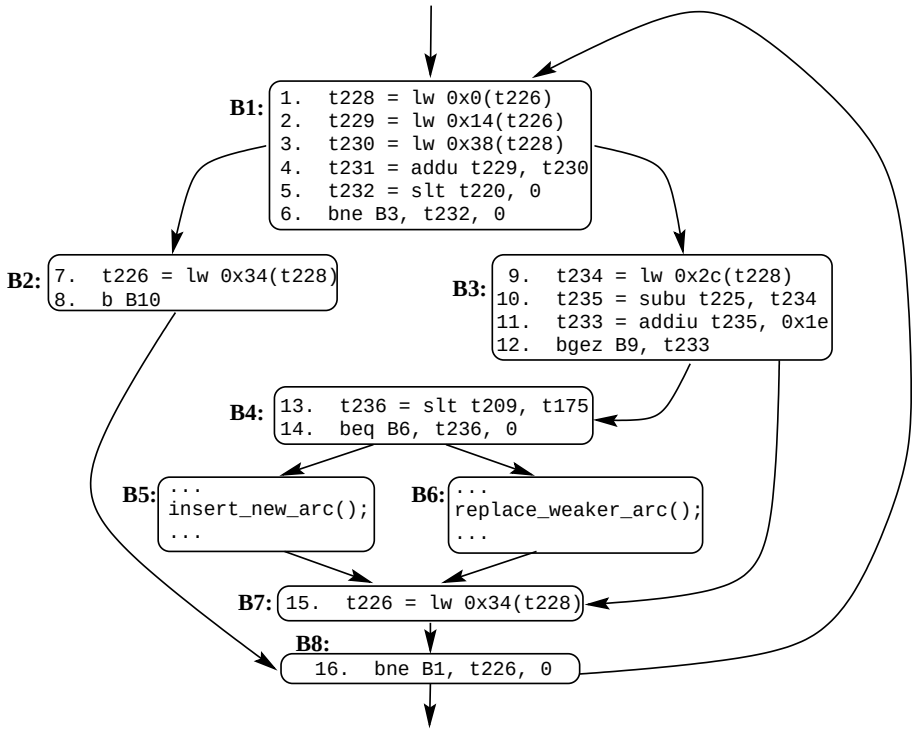

Figure 4: A portion of code taken from the **mcf** program

`= lw 0x14(t226)` in B1, are field loads.

For each induction pointer load our algorithm inserts an address update sequence and a prefetch instruction in basic blocks where the induction pointers are defined. Prefetch code for prefetching two iterations in advance of a load is shown in the Figure 5. The prefetch code is added for loads in lines 7 and 15 to B1, and for the load in line 1 to B2 and B7. The variables $t228s$ and $t226s$ are used to keep the values of the induction pointers $t228$ and $t226$ from the previous iteration.

To decide which loads should be prefetched, we need to analyze the *profitability* of prefetching each load. This profitability analysis, presented in the next section, takes into consideration the resource, memory, and recurrence constraints of the loop.

4 Profitability Analysis

Issuing prefetches at speculative addresses requires additional processor resources (e.g. issue slots) and additional memory bandwidth, because more memory operations are initiated in each loop iteration. The key intuition behind our profitability analysis is to balance the gain and the cost of prefetching. The gain comes from a reduction of the constraints imposed by recurrence

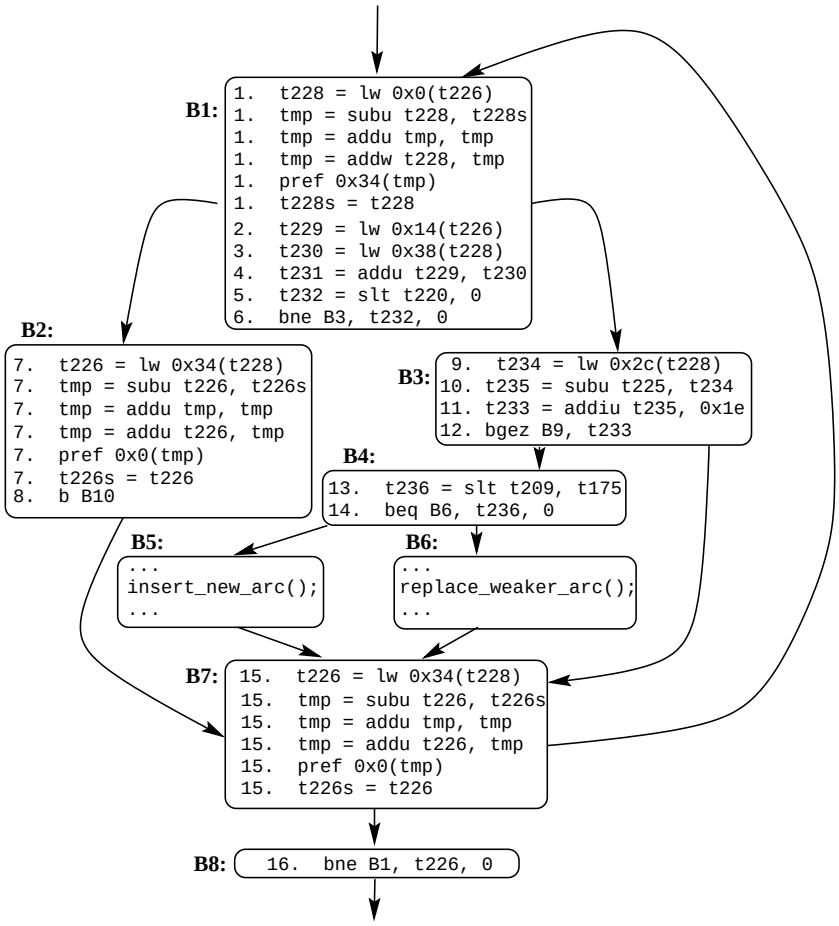

Figure 5: Prefetch code added to the **mcf** loop

cycles and the cost comes from the associated increase in processor and memory resource usage of the loop. In order to decide whether prefetching is profitable, we estimate lower bounds on the loop initiation interval with and without prefetching.

One lower bound can be derived from latency calculations around elementary circuits in the dependence graph. Each elementary circuit c, called a *recurrence*, is characterized by its *iteration distance*(c) and by its *latency*(c) [3]. The iteration distance of each dependence in c is equal to the number of iterations separating the dependent operations. The value of *iteration distance*(c) is the sum of the iteration distances along c. The value of *latency*(c) is the sum of the latencies of operations in c. The value of $rec_mii(c)$ for a recurrence c is:

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

Prefetching of an induction pointer load potentially reduces the $rec_mii(c)$ value of every recurrence c that contains this load by turning cache misses into cache hits. In the calculation

of rec_mii's the cache miss latency is initially used as the latency of loads. As prefetches are added, rec_mii is updated to reflect the fact that these loads become cache hits.

For a given basic block B we define the $res_mii(B)$ and $mem_mii(B)$, lower bounds on loop initiation interval imposed respectively by the resource and the memory requirements of the computation on control flow paths that execute B.

The usage of each resource in any control flow path p is calculated by adding the usage requirements of the operations in p. The $res_mii(p)$ is the maximum over all resources of the usage of the resource in p. For a given basic block B the $res_mii(B)$ is computed as:

$$res_mii(B) = \max_{p:B \in p} \{res_mii(p)\}$$

In other words, it is the maximum $res_mii(p)$ over all control flow paths p that execute B.

In order to estimate the impact of prefetching on the memory system we compute the lower bound imposed on the loop's initiation interval by the memory system capacity. The $mem_mii(p)$ of a control flow path p is computed from the estimated number of cache misses in p. Similar to $res_mii(B)$, the $mem_mii(B)$ is computed as:

$$mem_mii(B) = \max_{p:B \in p} \{mem_mii(p)\}$$

Because prefetch addresses are speculatively predicted, in estimating the number of cache misses we conservatively assume that every prefetch is a cache miss in addition to the miss of the original load. The estimate of the number of potential cache misses is based on a graph coloring algorithm.

For each basic block B to which the addition of the prefetch code is considered, we compute $res_mii^p(B)$ and $mem_mii^p(B)$, the resource and memory lower bounds on the initiation interval if the prefetch code were added to B.

The following condition states that the addition of prefetch code for a given induction pointer load L is profitable as long as the constraint imposed by the pointer-chasing recurrences that contain load L dominate constraints imposed by resource or memory usage on loop initiation interval in every of the control flow paths where the prefetch code is added:

Condition 1 (prefetch profitability condition) *Adding the prefetch code for an induction pointer load L to a basic block B is profitable if the following condition holds:*

$$max_{c:L \in c} rec_mii(c) > \max\{res_mii^p(B), mem_mii^p(B))$$

Consider adding prefetch for load 1 in our running example in Figure 3. This load's address is computed relative to induction pointer `t226`. `t226` is defined in two basic blocks, B2 and B7. Profitability analysis first computes the maximal $rec_mii(c)$ over all c's which contain the given load. Then the $res_mii(B)$ and the $mem_mii(B)$ of basic blocks B2 and B7 are computed.

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

addition of the address update sequence. In addition, prefetching of field pointers does not affect recurrences. It may still be beneficial to prefetch data for field loads for two reasons: first, with shorter load latencies registers are reused faster relieving register pressure, and second, fewer software pipeline stages are needed for overlapping loads with shorter latencies leading to faster loop prolog. Our algorithm estimates the amount of unused processor and memory resources in loop's control flow path. Prefetching of the field loads is considered profitable if there are enough unused processor and memory resources for such prefetching in every control flow path.

In the loop in our running example, after prefetches for the induction pointer loads `t226` and `t228` are added, our profitability analysis estimates that the memory bandwidth is fully utilized in every control flow path and no data is prefetched for any of the field loads (see Figure 5).

5 Prefetching Algorithm

An overview of our prefetching algorithm is shown in Figure 6. Fist we identify the set of pointer-chasing circuits C. Then we create a prioritized list L of the induction pointer loads and field loads. Loads are ordered according to their frequency of execution in the profile of the program. Induction pointer loads have higher priority than field loads. Each load operation o in L is initially marked as a cache miss. We perform a prefetch profitability analysis for each load operation o in L. If prefetching is deemed to be profitable for a given load, the compiler inserts the prefetch code. The priority ordering is important for two reasons: (1) data for a field load cannot be prefetched unless the address update sequence for the corresponding induction pointer has been instrumented by a previous prefetch for an induction pointer load; and (2) the insertion of the prefetch code for a load operation o_i might make prefetching for another load o_j that is analyzed later unprofitable.

```
DoPrefetch(P, E, v_i)
1.   C ← pointer-chasing recurrences
2.   L ← Prioritized list of load operations in C
3.   Mark each o in L as a cache miss
4.   for each o in L
5.        do if o is marked as cache miss and prefetch of o is profitable
6.             then add prefetch for o
7.                  mark o as cache hit
8.        endif
9.   endfor
```

Figure 6: Prefetching algorithm

5.1 Identification of List Traversing Circuits

We identify pointer-chasing circuits by inspection of the intermediate code. Figure 7 (a) and (c) display two examples of pointer-chasing circuit code in C. The corresponding assembly code generated by the compiler is shown in Figure 7 (b) and (d), respectively. Our key observation is that the circuit of loads with constant offset is indicative of a pointer-chasing circuit.[2]

`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)

Figure 7: Examples of code occurring in pointer-chasing loops

Conceptually our algorithm finds all elementary circuits in the loop's data dependence graph (for instance, using Tarjan's algorithm [15]) and then identifies the circuits that consist exclusively of copy instructions and loads with constant offsets. However, the number of circuits in the data dependence graph might be quite large, and in practice we do not need to enumerate all the circuits. This simplification has one important implication for the efficiency of the algorithm. The complexity of Tarjan's algorithm is $O(N\ E\ C)$, where N is the number of nodes and E the number of edges in the data dependence graph, and C is the number of elementary circuits. Our algorithm has a complexity of $O(N\ E\ P)$ where P is the number of pointer-chasing circuits. Since in practice there are very few pointer-chasing circuits in each loop, our implementation achieves a complexity of $O(N\ E)$ in practice.

5.2 Address Update Sequence

Although our experimental study indicated that often the memory access stride in linked lists is equal to the size of an element of the list, this is not always the case. For instance, if the linked list is interleaved with unrelated data in memory we might have a constant stride that is different from the element size. Moreover the same loop might use different strides when scanning different linked lists. The address update sequence dynamically calculates the memory stride of induction pointers. This sequence is specific to the target architecture. Our approach is robust with respect to changes in the stride between calls to the same function that contains the loop, as well as changes in the stride during a single operation of the loop, as long as these changes are not frequent.

The address update sequence is target dependent. The sequence for the MIPS R10000 microprocessor is shown in Figure 8. It consists of (1) a copy instruction to save the memory

[2]There are different ways in which a pointer-chasing loop can be translated in the intermediate form. Our approach does not identify other cases.

```
stride = addr - addr.prev
stride = stride * k
addr.pref = addr + stride
addr.prev = addr
pref addr.pref
```

Figure 8: Prefetch sequence for the R10000 processor

address currently being accessed; (2) a subtract instruction to calculate the memory access stride from one loop iteration to the next; (3) a shift/add instruction to multiply the stride by k in order to prefetch k iterations in advance, (4) an add instruction to compute the address that will be accessed two iterations from the current iteration, and (5) the prefetch instruction itself. In our implementation we experimented with $k = 1$ and $k = 2$. Therefore the second operation in the prefetch sequence is either unnecessary or can be implemented with a single add.

5.3 Computation of Res_mii(B)

For each basic block B, $res_mii(B)$, is computed using a DAG longest path algorithm [6] on the control flow graph. We associate a vector of weights with basic blocks in the control flow graph of the loop. Each element of the vector represents different processor resources (eg. register ports and functional unit stages) required by the corresponding basic block. We find the longest path for each resource among the control flow paths that execute B. The maximal such constraint over all resources gives the $res_mii(B)$.

5.4 Computation of Mem_mii(B)

For each basic block B, $mem_mii(B)$ is a function of the maximum number of cache misses that may occur on any of the control paths that execute B. Since it is not known at compile time whether a memory reference L will miss or hit in the cache, we use a coloring of the *cache miss interference graph* to estimate the number of cache misses in the loop.

Definition 1 *The cache miss interference graph is an undirected graph $G(N, E)$ where the set of nodes N represents memory operations in the loop, and the set of edges E represents the pairwise "interference" relationship between these memory operations. Two memory operations interfere if they do not fall into the same cache line.*

The interference relationship between two memory operations reflects the possibility that the two operations will simultaneously register cache misses. We assume that memory locations that are invariant over the loop iterations remain in the cache between iterations and are always cache hits. Therefore when constructing the interference graph, we mark these references as not interfering with any other memory references. Examples of references in this category include:

global-pointer-relative memory references, stack-pointer-relative memory references and loop invariant address memory references. We also assume that memory references on mutually exclusive control paths do not interfere with each other. Finally, if two memory references access memory using the same base address but different constant offset from the base, eg. two fields of one data structure, we assume that they do not interfere if the difference in their offsets is smaller than the cache line size.

When the cache miss interference graph is colored, two memory operations that may simultaneously miss in the cache are assigned distinct colors. Thus, the number of colors necessary to color such a graph corresponds to the maximum number of potential simultaneous cache misses along any control path of the loop. In general, the number of distinct colors necessary to color a node and all its adjacent nodes in the cache miss interference graph gives the maximum number of cache misses in one iteration along any control flow path that executes the corresponding memory reference.

The $mem_mii(B)$ for the basic block B in which a prefetch is being inserted is computed as follows:

$$mem_mii(B) = (N mod B_p + 1)L_p$$

where N is the number of distinct colors necessary to color L and all its adjacent nodes in the cache miss interference graph, B_p is the number of primary cache misses that can be served simultaneously by the memory system in question, and L_p is the delay caused by a primary cache miss.

6 Performance Evaluation

We have implemented our methods in the SGI MIPSpro C and Fortran compilers, layering them on state-of-the-art optimization and code generation technology. We compiled a set of the standard benchmarks, and executed the resulting code on an SGI Onix system, based on the MIPS R10000 microprocessor. The following performance evaluation is based on wall-clock measurements of execution time from those runs, compared against baseline measurements of the unmodified compiler. Our experimental results, in the follow subsections, evidence that

- The prefetching technique presented in this paper achieves significant speedup for some pointer intensive programs.
- Our profitability analysis is effective in balancing the reduction in number of cycles in a loop critical path and the increased processor resources and memory bandwidth requirements due to prefetching.

6.1 Experimental Framework

We performed our experiments on the Silicon Graphics Onyx machine with the 195 MHz MIPS R10000 processor, 32 KB primary and 1MB secondary caches. 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 could not 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 prefetching efficiency; second, an out-of-order processor can potentially reorder memory operations, and therefore affect the cache misses behavior. This may account for some small fraction of the execution time difference that can not be controlled by the compiler.

The experiments have been conducted under the IRIX 6.5 operating system with the machine running in a single user mode. We measured the wall-clock time for the execution of each benchmark.

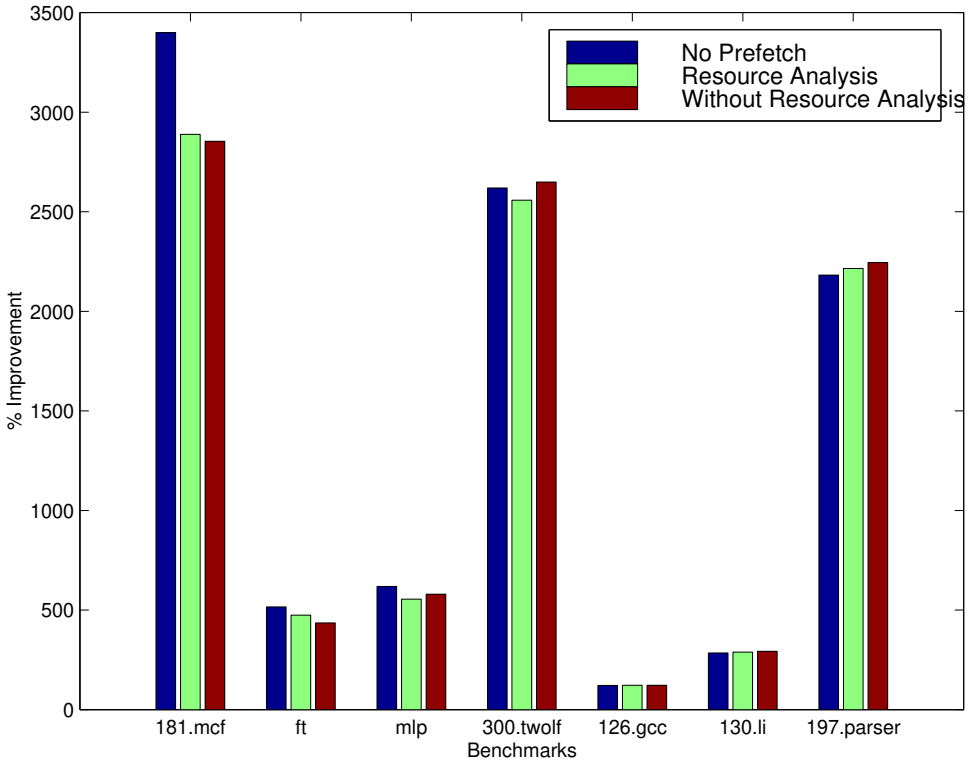

Figure 9: Execution time of benchmarks without prefetching, with three prefetch strategies

6.1.1 MIPSpro Compiler

The SGI MIPSpro compiler suite consists of highly-optimizing compilers for Fortran, C, and C++ on MIPS processors. It implements a broad range of optimizations, including interproce-

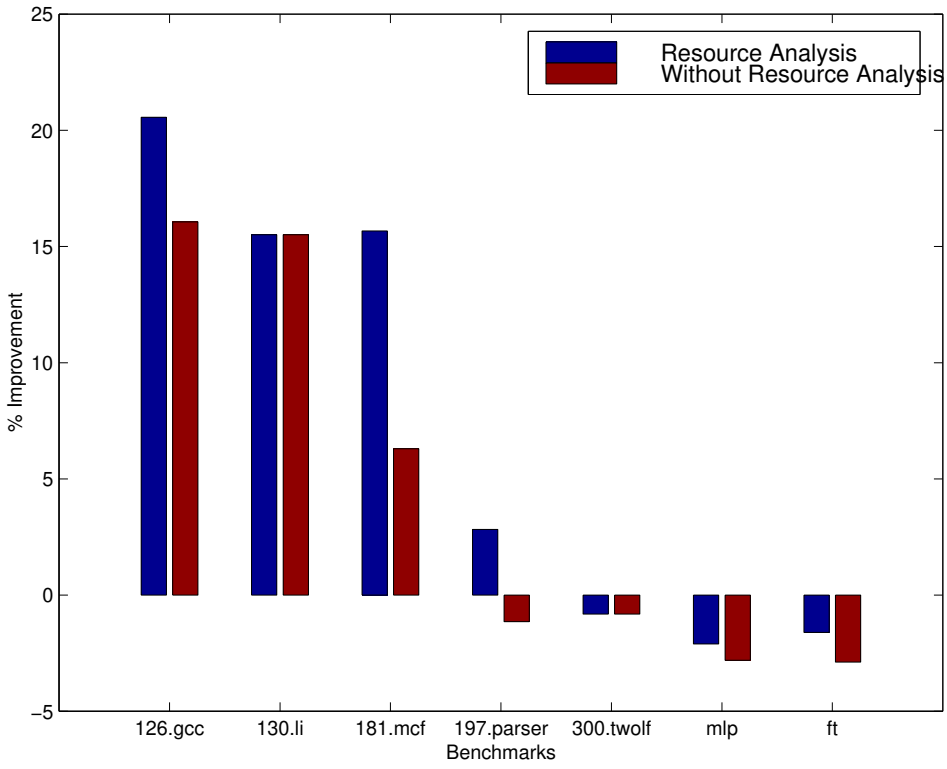

Figure 10: Percentage improvement of each strategy

dural analysis and optimization, loop nest rearrangement and parallelization, SSA-based global optimization, software pipelining, global and local scheduling, and global register allocation [8].

6.1.2 Test Programs

Benchmark	Benchmark Purpose
1. gcc	the GNU compiler
2. li	a lisp interpreter
3. mcf	a minimal cost flow program
4. parser	a syntactic parser of English
5. twolf	a place and route simulator
6. mlp	a receptron simulator
7. ft	a minimum spanning tree algorithm

Table 1: Test programs

For testing our prefetching technique, we used the set of programs in table 1. We chose these programs because they spend a significant portion of their execution time in loops traversing linked lists. Because our technique is designed to improve such loops, applying it to benchmarks that do not contain linked lists would be pointless.

6.2 Performance Measurements

The performance chart in Figure 9 plots the execution time of each program using three prefetch strategies: (1) without prefetching, (2) induction pointer prefetching with the profitability analysis, and (3) prefetching without profitability analysis. None of the test programs displays measurable performance degradation, and in three benchmarks the improvement is more than 15% in addition to all the standard optimizations of a industry-strength compiler.

The execution times shown in Figure 9 demonstrate that there is a significant improvement in the performance of mcf, ft, and mlp. The program that showed the greatest performance improvement is *mcf*. Our initial example loop takes up to 40% of this program execution time. It 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 this loop's initiation rate is constrained only by the pointer-chasing recurrence. The *mlp* 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 each neuron in the system, 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 sufficient so that prefetching makes a significant difference.

For twolf, gcc, li and parser, the difference between prefetching and not prefetching is less than 3%. Such a small difference in total execution time in an out-of-order processor might be caused by a number of factors. For instance, in the R10000, the instruction issue from the ready pool is pseudo-random, rather than FIFO, so the actual execution order (as opposed to graduation order) is somewhat random. Thus small changes in the order of the instructions and slight changes in the cache hit/miss pattern might cause 1-3% differences in wall-clock time measurements. Although *twolf* and *gcc* use linked lists extensively, the impact of prefetching is not visible because these linked lists are short and rarely suffer primary cache misses. *li* has a number of loops that operate on linked lists, where much of program execution time is spent. The input (SPEC95) we used only executes 1-2 iterations at a time - not enough to benefit from our prefetching. *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.

In order to evaluate the importance and effectiveness of our profitability analysis, we turned

it off in our implementation, i.e. 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. This caused performance loss in some of the programs. As the graph in Figure 10 shows, in two of the benchmarks, mcf and mlp, we observed more than 5% loss in improvement when prefetch is introduced indiscriminately in the code.

7 Related Work

Compiler prefetching for array-based numeric applications takes advantage of the data locality and of 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. On the other hand, in order for prefetching to be effective it is often necessary to speculatively predict addresses to drive prefetching. A number of prefetching techniques have been proposed that try to address these problems.

In [16], use of profiling information is suggested for identification of potential cache misses. A compiler analysis combined with instruction scheduling is developed in [13] based on the observation that certain kinds of memory references that represent small fraction of static instructions tend to be responsible for the majority of cache misses. Finally, the authors of [11] use the fact that often the data at the address passed as a parameter to a function call suffer the miss, and they issue prefetch for such cases.

Address speculation has been typically used in hardware prefetching schemes. They exploit the fact that often the addresses referenced by loads and stores follow an arithmetic progression, i.e. the effective address of a given memory reference is equal to the previous address of the same static instruction plus a constant stride. By keeping track of the last effective address and of the address stride, previously unseen addresses are speculatively predicted [9, 5, 10]. In particular, Selvidge [7] and Mehrotra [14] noticed the regularity in memory streams generated by the linked lists accesses. A similar hardware approach is to reproduce the address generation process in hardware and to perform it in advance of other computations [1]. The difficulty in applying such techniques is that since the hardware does not have knowledge about the program, it is very difficult to differentiate between memory references. The hardware is forced to track the memory footprint of all the references, while only some of them may possess regularity. On the other hand, because of the limitations on the number of references that can be entered in the history tables, entries may interfere reducing the efficiency of prefetching.

Several software prefetching methods use address speculation [4, 2]. The main idea is to remember the history of the memory access pattern to a given data structure and to follow this pattern for predicting future addresses. In contrast to the hardware prefetching, software prefetching can use program knowledge to place prefetches where they will do the most good, and avoid issuing them when redundant. It may also be able to exploit the application specific knowledge about memory access patterns. However, such prefetching suffers from two main

pitfalls: firstly, the overhead associated with additional prefetch instructions in the instruction stream; and secondly, it is often difficult to predict the run-time behavior of a given static memory reference at compile time.

8 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 prefetching 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. For some benchmarks, we obtained gains of performance of 15-20% over their performance after standard compiler optimizations.

References

[1] 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.

[2] A.Roth and G.Sohi. Effective jump-pointer prefetching for linked data structures. In *International Symposium on Computer Architecture*, pages 111 – 121, 1999.

[3] B.Rau. Iterative modulo scheduling. Technical Report HPL-94-115, HP Laboratories, 1995.

[4] 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.

[5] 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.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[7] C.Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance.* PhD thesis, MIT, 1992.

[8] 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.

[9] J. Fu and J. Patel. Stride directed prefetching in scalar processors. In *International Symposium on Microarchitecture*, pages 102 – 110, 1992.

[10] J.Gonzales and A.Gonzales. Speculative execution via address prediction and data prefetching. In *International Conference on Supercomputing*, pages 196 – 203, 1997.

[11] 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.

[12] T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, 1994.

[13] 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.

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

[15] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211 – 216, September 1973.

[16] 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.