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

Automatic Prefetching of Induction Pointers

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

CAPSL Technical Memo 38
April 18, 2000

†Stoutchinin is with STMicroelectronics, Jim Dehnert is with Silicon Graphics Inc, Suneel Jain is with Narus Inc.
This is a revision that replaces CAPSL Technical Memo 37 of November 1999.

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

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 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 regressions.

Contents

List of Figures

List of Tables

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. 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.

```
max = 0;
current = head;
while(current != NULL) {
  if(current -> key > max)
    max = current -> key;
  current = current -> next;
 }
```

```
max = 0;
current = head;
tmp = current;
while(current != NULL) {
  if(current -> key > max)
    max = current -> key;
  current = current -> next;
  stride = current - tmp;
  prefetch(current + stride*k);
  tmp = current;
 }
```

Figure 1: A *pointer-chasing* loop that scans a linked list without and with prefetching.

Prefetching in conjunction with software pipelining has been successfully used in optimization of numerical loops [9]. 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 prefetching insertion. We use statements in C language in Figure 1 for easy understanding. In Section 3 we show the actual intermediate code 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 compute an estimation for the number of outstanding cache misses in any iteration of the loop. We use this prediction to avoid prefetching when the resulting number of cache misses would exceed the maximum number of outstanding misses supported by the processor.

We 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 35% and very little performance degradation in some benchmarks. The improvements of 15 to 35% in the execution time is over the fully optimized code.

We present an overview of our prefetch framework through an example in section 3. We describe our profitability analysis in section 4 and the prefetch algorithm in section 5. 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 6. Finally we discuss work related with this research in section 7.

2 Problem Statement and Open Questions

The research problem addressed in this paper can be posed as the following set of questions:

- How to identify pointer-chasing recurrences using a low complexity algorithm? (see Section 4.1)
- How to decide when there are enough processor resources and available memory bandwidth to profitably prefetch an induction pointer? (see Section 4.5)
- 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 5)
- 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 4.4)
- How well does speculative prefetching of induction pointers works on an industry-strong state-of-the-art optimizing compiler? (see Section 6)

3 Overview of the Approach and the Running Example

In this section we present an overview of our prefetch method and discuss its application to a pointer-chasing loop from the **mcf** program[1]. We present the source code for the most important loop of **mcf** in Figure 2. The intermediate code generated by the original compiler for this loop is shown in Figure 3. The corresponding prefetch code generated by the compiler modified with our framework is shown on Figure 4. 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 2. This loop is also responsible for nearly half of the L1 cache misses in the program. As shown in Section 6 prefetching induction pointers in the mcf program resulted in a performance improvement of 20%!

The loop in Figure 2 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). Through instrumentation of the code, we observed that the memory accesses for `arcin` and `tail` had

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

```
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 2: A portion of code taken from the **mcf** program.

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 3 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 on the load in line 1. `t226` and `t228` are, thus, induction pointers and the loads in lines 1, 7, and 15 above are *induction pointer* loads. The rest of the loads whose address is calculated relative to `t226` and `t228`, for example load `t229 = lw 0x14(t226)` in B1, are *field loads.*

To prefetch an 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 on Figure 4. 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. Observe that we may also prefetch field loads. Because the address referenced by a field load differs from the corresponding induction pointer by a constant offset, the address computed by the update sequence for an induction pointer can also be used for the prefetching of field loads.

Observe that because the stride is computed dynamically at runtime, this prefetch technique is robust to infrequent changes in stride, or to occasional deletion/addition of elements to an

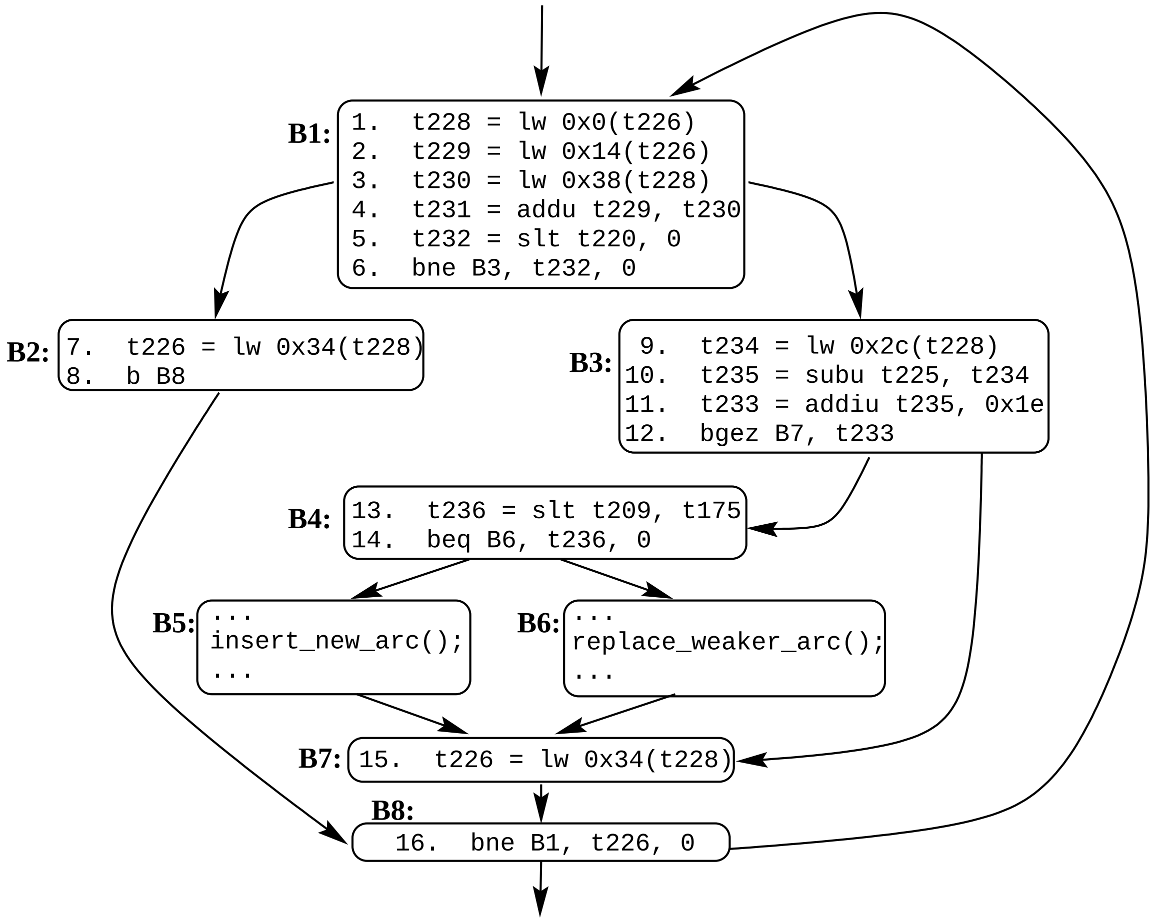

Figure 3: The original compiler generated code (before prefetching) for the portion of code taken from the **mcf** program.

existing long recursive data structure.

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 Induction Pointer Identification and Profitability Analysis

In our framework the issue of a prefetch 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 the program can have the following consequences:

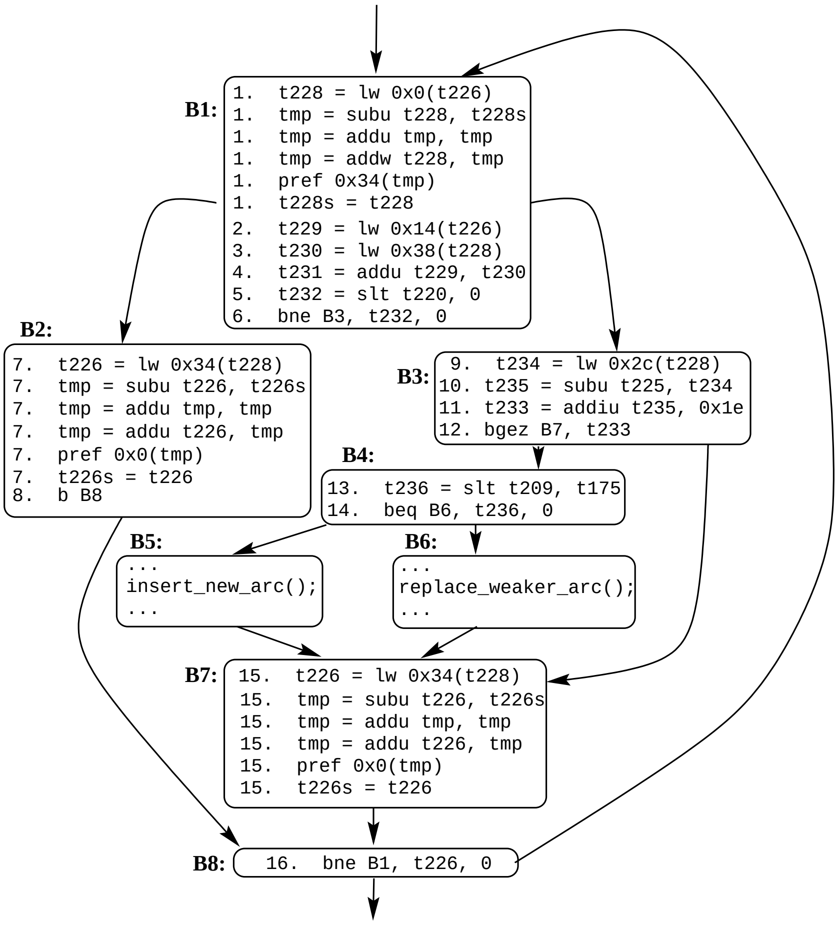

Figure 4: The code generated by the modified compiler after prefetching for portion of code taken from the **mcf** program.

1. There are more instructions to issue consuming additional processor resources.[2]

2. There is more memory traffic to fetch the instructions, and potentially additional data traffic (in case of misprediction).

3. The longer code might cause disruption in the instruction cache.

4. Mispredicted prefetching might displace potentially good data from the cache.

Our profitability analysis ameliorate consequences 1 and 2 above. In our experience pointer chasing loops are short, thus consequence 3 is not an important concern. In our current implementation we do not address the consequence 4.

[2]In this discussion "processor resources" refers to address computation units, issue slots, register file ports, etc.

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. We estimate the constraints imposed by processor resources and by recurrent data dependencies on the initiation interval of the loop to decide when we should prefetch for a given induction pointer.

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 estimations: (1) a conservative bound on its initiation interval; (2) an estimation of the increase in cache misses that may occur when we prefetch for L; and (3) an estimation of the resource usage in the control paths that execute L. Prefetching of L is deemed profitable only if it reduces the initiation interval of L and if a potential extra cache miss caused by the prefetch does not result in a stall of the cache fetch mechanism.

4.1 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 5 (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 5 (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.

`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 5: Examples of code occurring in *pointer-chasing* loops

We used the two patterns shown in Figure 5 for identification of the pointer-chasing circuits. Conceptually our algorithm finds all the elementary circuits in loop's data dependence graph using Tarjan's algorithm [17] and then finds those circuits consisting exclusively of loads with constant offsets and copy instructions. In our implementation, we do not enumerate all the circuits in the data dependence graph (that can be exponential in a general graph) but identify 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 edges in the data dependence graph, and C is the number of elementary circuits. Since in practice the number of the pointer-chasing circuits in a loop is very few (one most of the time), our implementation practically achieves $O(N * E)$ complexity.

4.2 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 over the recMII for all the elementary circuits in the data dependency graph of the loop.

In this framework we are 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 an operation of L participates. Therefore $recMII(B)$ is the maximum of $recMII(c)$ among all recurrences that execute L. The $recMII(L)$ defines a *conservative bound* on the initiation interval of the 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 recomputed. Therefore prefetching a load L reduces the $recMII(c)$ for all the recurrences that contain the load L.

4.3 Resource minimum initiation interval

The resource usage by 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). [3] 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.

4.4 Cache/memory available bandwidth

If an instruction that prefetches data at a mispredicted address is added to a control path p, it may generate an extra cache miss in that path. If this extra cache miss causes the processor to stall, the prefetching is counter-productive. However, in processors with non-blocking caches the processor can generate up to k memory references that miss the cache before it stalls. Therefore, for such machines, we define the following *Outstanding Miss Optimization* (OMO) problem:

> *Optimize prefetching by making the best use of the available bandwidth between the cache and the memory.*

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. Like with processor resources, we use a conservative estimation of available memory bandwidth. $M(B)$ is defined as the minimal available bandwidth of 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 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: $M(p) = k - m(p)$.

[3] We do not consider the loop's back edge when applying the longest path algorithm to the CFG.

4.5 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:*

$$resMII^P(B) \leq recMII^P(L) \wedge S(B) > 0$$

where $resMII^P(B)$ and $recMII^P(L)$ are the resource MII for the basic block B and the recurrence MII for the load L after the prefetch sequence for L 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 after prefetching the initiation interval of the loop is still limited by 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 right 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.

Consider the block $B1$ in our example in Figure 3. This basic block belongs to two data recurrences formed by the statements $1 \rightarrow 15 \rightarrow 1$, and $1 \rightarrow 7 \rightarrow 1$. The iteration distance for both recurrences is one, thus the $recMII(B1)$ is given by $lat(1) + max(lat(7), lat(15))$. Block $B1$ belongs to four distinct control paths. The processor resource usage along each of these paths after the insertion of the prefetch sequence is computed, and the maximum usage over the four control paths for that resource is taken. The most constraining resource is then selected to determine $resMII^P(B1)$.

4.6 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. 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. 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.

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 4).

5 Prefetch Algorithm

Now that we have defined the recurrence MII for operations, the resource MII, and the available memory bandwidth for a basic block in a loop that contains complex control flow, we can present the outline of our prefetch algorithm. The goal of the 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.

```
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 o in O, o ∈ B
7.      do if recMII^P(o) ≤ resMII^P(B) and S(B) > 0
8.            then add prefetch for o to B;
9.                 mark o as cache hit;
10.        endif
11. endfor
```

Figure 6: Prefetch algorithm.

The first step in our prefetch algorithm is to identify the pointer chasing recurrences and to create a prioritized list of the load operations in these recurrences. In order to identify the pointer-chasing recurrences, we build the data dependence graph of the loop. The existence of a recurrence manifests itself as a circuit in the dependence graph. We define *pointer-chasing recurrences* as dependence graph circuits that involve only loads or copy operations. The loads in recurrences are prioritized (step 2-4 in Figure 6) according to the frequency of execution of the basic blocks to which they belong. Loads that are not part of a recurrence, and are not loop invariant, relative to global pointer, or relative to the stack pointer are also candidates for prefetching. They are added to the N list in step 3, which 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. We will prefetch for the loads in N if there are enough resources available after we prefetch for the loads in pointer chasing recurrences. 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. [4]

For each load operation in the priority list, $o \in B$, 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 basic block B, $recMII^P(B)$, as the maximum of the recMII of all recurrences to which B 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 $recMII^P(B) \geq resMII^P(B)$, 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 o as a cache hit for future memory bandwidth availability estimation, otherwise remove the prefetch sequence.

The use of the most constrained recurrence and the most constrained control path to estimate $recMII^P(B)$ 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 an operation o 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 recurrence to which o belongs. Therefore these quantities must be computed for each operation that we consider for prefetching.

5.1 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 chromatic number of the miss interference graph, i.e. the minimum number of colors necessary to color the graph, corresponds to the maximum

[4] 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.

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.

For a basic block B, the number of distinct colors necessary to color all the loads in B and their adjacent nodes in the cache miss interference graph gives the estimate on the maximum number of cache misses to be expected along any control path that executes B. Therefore the available memory bandwidth of B, $M(B)$ can be computed as follows:

$$M(B) = k - N$$

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.

5.2 Prefetch Sequence

The goals of the prefetch sequence are to track the effective memory addresses of the loads involved in the pointer-chasing recurrences and to compute the memory access strides for such loads. A more efficient way of achieving the same goal would be to have a special load instruction in the ISA that performs the address tracking and the stride computation as part of its hardware specification.

The prefetch sequence in the MIPS R10000 machine consists of (1) a copy instruction to save the memory address currently accessed; (2) a subtract instruction to calculate the memory access stride from one loop iteration to the next; (3) a multiply instruction to multiply the stride by the number of iterations in advance that we want to prefetch, (4) an add instruction to compute the address that will be accessed two iterations from the current iteration, and (5) the prefetch instruction itself.

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

Figure 7: Prefetch sequence for the R10000 processor.

The prefetch sequence is target dependent. On the R10000 processor after the load address, addr, is computed, we insert the sequence shown in Figure 7. In this prefetch sequence, k is the number of iterations we want to prefetch in advance. 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 left shift.

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 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 prefetch technique presented in this paper achieves significant speedup for programs that spend significant time on pointer chasing loops (see Table 2.
- A static analysis of benchmarks indicates that a significant number of programs — beyond the benchmarks studied in this paper will benefit from pointer prefetching (see Table 5).
- 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 2 and 3).
- 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 (see Section 6.4). For most of these programs, the profitability analysis prevents prefetching from resulting in performance degradation (see Table 3).

6.1 Experimental Framework

We performed our experiments on a Silicon Graphics Onyx machine with 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 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 prefetch efficiency; second, an out-of-order processor can reorder memory operations, and therefore affect the cache miss behavior.

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 with the machine running on 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. An improvement in the estimation of the available memory bandwidth reduced the regressions compared with the data in an earlier version of this paper [11].

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 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].

6.1.2 Test programs

For testing our prefetch technique, we used the set of programs in table 1. For the benchmarks in the SPEC2000 benchmark suite we used the reference input. 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 When pointer prefetching works

Table 2 shows four 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) induction pointer prefetching with the profitability analysis, and (3) prefetching without profitability analysis. In the 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 improve-

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 perceptron simulator
7. ft	a minimum spanning tree algorithm

Table 1: Test Programs.

Benchmark	Execution Time			Performance Improvement	
	No Prefetch	No Analysis	With Analysis	No Analysis	With 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 %
130.twolf	2.540 sec.	2.657 sec.	2.531 sec.	-4.6 %	0.4 %

Table 2: Execution time in seconds; performance improvement for programs that benefit from prefetching.

ments of more than 15% (with a remarkable 35% for ft) on top of all the standard optimizations of a industry-strength compiler. The profitability analysis transforms a performance regression of 4.6% into a performance improvement of 0.4% for 130.twolf.

The execution times shown in Table 2 demonstrate that there is a significant improvement in the performance of mcf, ft, and mlp. The loop presented in Appendix 3 takes up to 40% of the mcf program execution time. That 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 (see Table 4). *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 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.

6.3 When pointer prefetching does not help

Benchmark	Execution Time			Performance Regression	
	No Prefetch	No Analysis	With Analysis	No Analysis	With Analysis
126.gcc	122 sec.	123 sec.	122 sec.	0.7 %	0.1 %
130.li	285 sec.	293 sec.	292 sec.	2.8 %	2.5 %
197.parser	2.180 sec.	2.245 sec.	2.189 sec.	3.0 %	0.5 %

Table 3: Execution time in seconds; performance regression for programs that do not benefit from prefetching.

Table 3 displays the results for three 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 regression caused by prefetching. As the table shows the profitability analysis is effective on the reduction of this regression for gcc and parser.

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. In the case of gcc, prefetching disturbs the virtual memory mechanism and causes a significant increase in the number of TLB misses (see Table 4). 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 regression in li, and the 20% increase in the number of cache misses when prefetch is inserted (see Table 4) indicates that li might not have stride regularity, and that mispredicted addresses for prefetching are displacing useful data in 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 when our profitability analysis is used the performance regression for parser is only 0.5%.

6.4 Cache misses and TLB misses

Table 4 shows measurements of L1 cache misses and TLB misses obtained using the `perfex` tool to read the processor's hardware counters. The first two columns show the L1 cache hit rate for the entire program when no prefetch is applied and with prefetching. Notice that the programs that benefit the most from pointer prefetching (mcf, ft, and mlp) are the ones that had a low hit rate when no prefetch was applied. The two columns to to the right of Table 4 show the percentage variance in the number of L1 cache misses and TLB misses between the execution of the program with and without prefetching. A negative percentage indicates that the number of misses decreased when prefetching was applied, while a positive percentage indicates an increase in the number of misses. This data indicates that for the benchmarks that do not benefit from

Benchmark	L1 Hit Rate		Increase in	Increase in
	No prefetch	Pref.+Analysis	L1 misses	TLB misses
181.mcf	0.665	0.714	0.3 %	-0.1 %
ft	0.383	0.606	3.1 %	-0.6 %
mlp	0.807	0.832	0.5 %	14.0 %
130.twolf	0.917	0.917	-0.1 %	-9.7 %
126.gcc	0.984	0.983	6.7 %	22.0 %
130.li	0.988	0.986	20.0 %	2.0 %
197.parser	0.963	0.961	7.6 %	7.0 %

Table 4: L1 cache hit rate for the versions without prefetching and with prefetching and analysis, increase in the number of L1 misses and in the number of TLB misses.

pointer prefetching, there is either an increase in the number of L1 cache misses, an increase in the number of TLB misses, or both. These changes in L1 cache misses and TLB misses can be explained by the displacement of data from these structures because of the insertion of prefetching into the stream of memory accesses. An alternative explanation is that the recursive data structures in those benchmarks have irregular strides. We are currently implementing a "stride regularity profiling" technique to identify which recurrences are good candidates for our prefetch technique. We expect that with this profiling mechanism we will eliminate the minor performance regressions that appear in Table 3.

6.5 Applicability of pointer prefetching

An interesting question to be answered is how often the technique presented in this paper is applicable. We have an indication that the technique applies quite often. We statically measured the number of pointer chasing loops for the benchmarks in Table 5. In this table, loops is the number of loops analyzed by our framework in that benchmark, pointer chasing loops (PCL) is the number of loops where we found pointer chasing recurrences, and prefetched PCL is the number of loops in which our profitability analysis indicated that it is profitable to prefetch.

The graphs in Figure 8 summarize the performance results for all benchmarks that we studied. Figure 8(a) plots the execution time while Figure 8(b) plots a percentage performance improvement.

7 Related Work

Compiler prefetching for array-based numeric applications takes advantage of the data locality and of the compiler ability to analyze numerical loops [9]. Application of prefetching to dynamically allocated irregular data structures is more difficult. Such data structures can not be

benchmark	loops	PCL	Prefetched PCL	Source
li	107	37	28	SPEC95
ft	19	10	5	SPEC95
mlp	63	38	22	application
gcc	1921	755	555	SPEC2000
parser	412	215	94	SPEC2000
twolf	668	231	146	SPEC2000
mcf	45	13	12	SPEC2000
jkbench	23	3	2	application
gzip	153	0	0	SPEC2000
vpr	296	19	5	SPEC2000
crafty	276	0	0	SPEC2000
perlbmk	627	69	37	SPEC2000
gap	1015	7	3	SPEC2000
vortex	175	1	0	SPEC2000
bzip2	112	0	0	SPEC2000

Table 5: Number of loops with pointer chasing loops and the number that the profitability analysis decides to prefetch.

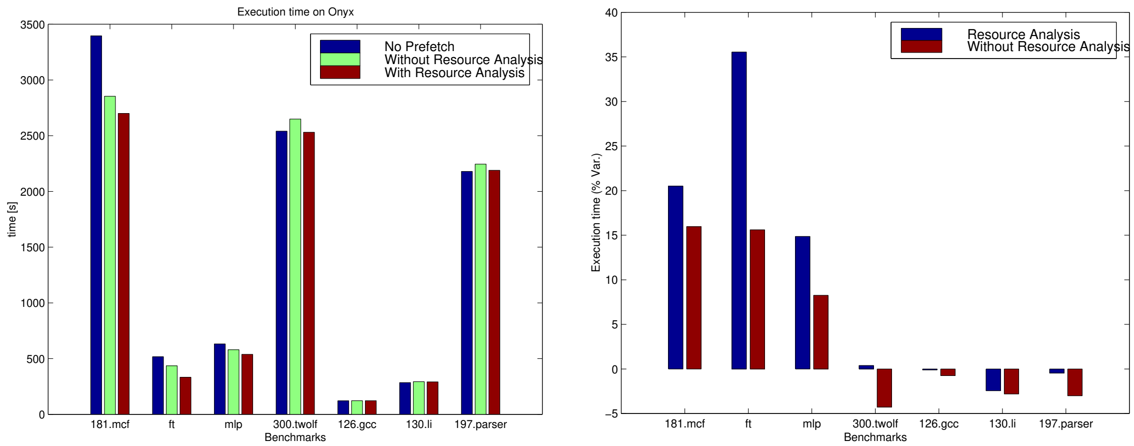

Figure 8: (a)Execution time of benchmarks without prefetching, with prefetching without performance analysis; and with prefetching and performance analysis; (b) Percentage improvement of each strategy;

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. A number of prefetching techniques have been proposed that try to address these problems.

Mowry and Luk use profiling information to identify potential cache misses [10]. Ozawa *et al.* develop a compiler analysis combined with instruction scheduling 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 [12]. 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 [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, 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 [1, 4, 5]. In particular, Selvidge [16] and Mehrotra [8] 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]. The difficulty in applying such techniques is that since the hardware does not have knowledge about the program, it is very difficult to distinguish different types of memory references. The hardware is forced to track the memory footprint of all references, when only some of them may follow a regular pattern. On the other hand, because the size of a hardware history table is necessarily limited, interference between table entries reduces the efficiency of prefetching.

Several software prefetching methods use address speculation [7, 15]. 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 compile time analysis to place prefetches where they will do the most good, and avoid issuing prefetches when they are redundant. Software prefetching may also exploit the application specific knowledge about the memory access patterns. However, software prefetching suffers from the inability to predict, at compile time, the run-time behavior of a dinamically allocated memory references.

Our approach is different from the one proposed by Luk and Mowry both on the identification of pointer chasing loops and on 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 a 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.

Luk and Mowry propose the creation of an extra field in the data structure to store the address that will be actually fetched k iterations in advance. Therefore the implementation of prefetching in their framework requires profiling of the code to compute these addresses, extra storage space, and the execution of an actual load for every induction pointer prefetch. In contrast our framework only requires a few address computation instructions. The advantage of

Luk and Mowry's approach is that the actual address to be prefetched is known, and therefore always correct. We, on the other hand, speculate on the regularity of the data structure stride and might prefetch the wrong address sometimes. To compensate miss predictions we developed the profitability analysis to avoid prefetching when the processor resources are scarce.

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 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 of performance of 15-35% over their performance after standard compiler optimizations, while displaying minimal performance regression for others. This simplicity — a small set of extra instruction for dynamic address calculation, no reallocation of data, no hardware support, and currently not even profiling — allowed for an effective implementation in an industry-strong compiler. Moreover, there was no noticeable change in the time required to compile the benchmarks that we tested when the prefetch analysis was executed.

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] Thomas H. Cormen, Charles E. Leiserson, and Ronald 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. Mehrotra. *Data Prefetch Mechanisms for Accelerating Symbolic and Numeric Computation.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.

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

[10] 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.

[11] ommited. Automatic prefetching of induction pointers for software pipelining. Technical Report, November 1999.

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

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

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