Compiler-Based Prefetching for Recursive Data Structures

Chi-Keung Luk and Todd C. Mowry

Department of Computer Science
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 3G4
{luk,tcm}@eecg.toronto.edu

Abstract

Software-controlled data prefetching offers the potential for bridging the ever-increasing speed gap between the memory subsystem and today's high-performance processors. While prefetching has enjoyed considerable success in array-based numeric codes, its potential in pointer-based applications has remained largely unexplored. This paper investigates compiler-based prefetching for pointer-based applications—in particular, those containing recursive data structures. We identify the fundamental problem in prefetching pointer-based data structures and propose a guideline for devising successful prefetching schemes. Based on this guideline, we design three prefetching schemes, we automate the most widely applicable scheme (*greedy prefetching*) in an optimizing research compiler, and we evaluate the performance of all three schemes on a modern superscalar processor similar to the MIPS R10000. Our results demonstrate that compiler-inserted prefetching can significantly improve the execution speed of pointer-based codes—as much as 45% for the applications we study. In addition, the more sophisticated algorithms (which we currently perform by hand, but which might be implemented in future compilers) can improve performance by as much as twofold. Compared with the only other compiler-based pointer prefetching scheme in the literature, our algorithms offer substantially better performance by avoiding unnecessary overhead and hiding more latency.

1 Introduction

Memory latency is becoming an increasingly important performance bottleneck as the gap between processor and memory speeds continues to grow. While cache hierarchies are an important step toward addressing the latency problem, they are not a complete solution. To further reduce or tolerate memory latency, automatic compiler techniques such as locality optimizations [4, 21] and software-controlled prefetching [3, 16] have been proposed and evaluated in the past. While these techniques have shown considerable promise, they have been limited in scope to array-based numeric applications. In this paper, we explore how to expand the compiler's scope to include another important class of applications: those containing pointer-based data structures (also known as "recursive" data structures).

Recursive **D**ata **S**tructures (**RDS**s) include familiar objects such as linked lists, trees, graphs, etc., where individual nodes are dynamically allocated from the heap, and nodes are linked together through pointers to form the overall structure. For our purposes, "recursive data structures" can be broadly interpreted to include most pointer-linked data structures (e.g., mutually-recursive data structures, or even a graph of heterogeneous objects). From a memory performance perspective, these pointer-based data structures are expected to be an important concern for the following reasons. For an application to suffer a large memory penalty due to data replacement misses, it typically must have a large data set relative to the cache size. Aside from multi-dimensional arrays, recursive data structures are one of the most common and convenient methods of building large data structures (e.g, B-trees in database applications, octrees in graphics applications, etc.). As we traverse a large RDS, we may potentially visit enough intervening nodes to displace a given node from the cache before it is revisited; hence temporal locality may be poor. Finally, in contrast with arrays, where consecutive elements are at contiguous addresses and therefore stride-one accesses can exploit long cache lines, there is little inherent spatial locality between consecutively-accessed nodes in an RDS since they are dynamically allocated from the heap and can have arbitrary addresses. Therefore, techniques for coping with the latency of accessing these pointer-based data structures are essential.

1.1 Coping with Memory Latency for RDSs

Ideally, the first step toward coping with memory latency would be to *reduce* latency by restructuring either the computation or the data to minimize cache misses. Unfortunately, the locality optimizations developed for numeric applications (e.g., tiling, loop interchange, etc. [4, 21]) are not applicable to RDSs. Although we do explore one optimization which potentially improves spatial locality in RDSs (*data linearization*, as described later in Section 2.2.3), a significant number of cache misses still remain, and therefore techniques for *tolerating* latency are also needed.

Tolerating write latency is not a fundamental problem, since we can buffer and pipeline writes. The real challenge is tolerating *read* latency, which requires that we decouple the *request* for data from the *use* of that data, while finding enough useful parallelism to keep the processor busy in between. The two main techniques for tolerating read latency are *prefetching* [2, 3, 15] and *multithreading* [1, 9, 11, 13]. Prefetching tolerates latency by anticipating what data is needed and moving it to the cache

0

ahead of time. Prefetching can be controlled either by hardware or software. Hardware-based schemes typically look for patterns in previous accesses to predict future behavior, and software-based schemes rely on either the programmer or the compiler to insert explicit prefetch instructions. In contrast, multithreading tolerates latency by switching from one concurrent thread to another upon a cache miss.

Comparing prefetching with multithreading, the advantage of multithreading is that unlike prefetching, it does not require the ability to predict data addresses ahead of time. Although this is not an issue in numeric codes (since affine array addresses can always be computed ahead of time), it does present a serious challenge when we want to prefetch RDS codes due to the tight control and data dependences (we will discuss this in detail later in Section 2). Prefetching, however, does offer some important advantages over multithreading. First, prefetching can accelerate a *single* thread of execution, unlike multithreading which requires multiple concurrent threads. This additional concurrency may or may not exist—particularly in a uniprocessor environment, it is unlikely that the programmer would go through the pain of parallelizing their application just for the sake of multithreading. Second, multithreading requires a significant amount of hardware support to minimize thread switching overhead (e.g., replicated register files). In contrast, software-controlled prefetching requires very little hardware support—in fact, most commercial microprocessors today already support prefetch instructions.

Comparing software vs. hardware-controlled prefetching, a potential drawback of a software-based approach is the additional instruction overhead necessary to issue prefetches. However, with today's superscalar processors, it may be possible to overlap much of this overhead with other computation. The software-based approach has the major advantages of requiring less hardware support (the hardware already exists), being more flexible, and (most importantly) being able to exploit application-specific knowledge about future access patterns. (It is not clear how a hardware-based scheme could predict the chaotic-seeming access patterns that occur when an RDS is traversed.)

Up until now, there has been no suitable solution for tolerating the latency of RDS codes in commercial systems: there are no commodity microprocessors that support multithreading, effective hardware-based prefetching schemes have yet to be demonstrated, and the compiler support for exploiting software-controlled prefetching for RDSs has remained an open question. In this paper, we address this open research question by designing and evaluating compiler-based prefetching schemes which successfully tolerate the latency of accessing recursive data structures in modern microprocessor-based systems.

1.2 An Overview

This paper is organized as follows. We begin in Section 2 by identifying the fundamental problem that makes prefetching RDSs difficult, and proposing a guideline for devising successful prefetching schemes. Based on this guideline, we design three different prefetching algorithms. In Section 3, we describe how the most widely applicable of these schemes (*greedy prefetching*) has been implemented in an optimizing research compiler (SUIF). Section 4 describes our experimental framework, and Section 5 presents our experimental results where we evaluate all three prefetching algorithms on the Olden benchmark suite through detailed simulations of a MIPS R10000-like processor. Section 6 discusses related work, including a quantitative comparison between our schemes and the only other compiler-based pointer prefetching scheme proposed in the literature. Section 7 describes future work to improve our algorithms, and finally, we conclude in Section 8.

2 Software-Controlled Prefetching for RDSs

In this section we discuss the major issues and challenges involved in software-controlled prefetching for RDSs, we present guidelines for overcoming these challenges, and we describe three prefetching algorithms based on these guidelines.

2.1 Challenges in Prefetching RDSs

Any software-controlled prefetching scheme can be viewed as having two major phases. First, an *analysis* phase predicts which dynamic memory references are likely to suffer caches misses, and hence should be prefetched. Second, a *scheduling* phase attempts to insert prefetches sufficiently far in advance such that latency is effectively hidden, while introducing minimal runtime overhead. For array-based applications, the compiler can use *locality analysis* to predict which dynamic references to prefetch, and *loop splitting* and *software pipelining* to schedule prefetches [15].

A fundamental difference between array references and pointer dereferences is the way addresses are generated. The address of an array reference $A[i]$ can always be computed once a value of i is chosen. In contrast, the address of a pointer dereference $*p$ is unknown unless the value stored in p is read. This difference makes both the analysis and scheduling phases significantly more challenging for RDSs than for arrays.

(a) Example Code

```
for (i=1; i<=n; i++) {
  listNode *p = listHead[i];
  while(p) {
    work(p→data);
    p = p→next;
  }
}
```

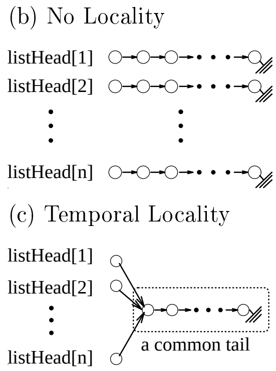
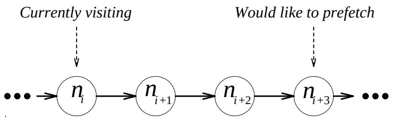


Figure 1: Example of list traversals, both with and without temporal data locality.

2.1.1 Analysis

To illustrate the difficulty of analyzing data locality in RDSs, consider the code in Figure 1(a), where we are traversing `n` different linked lists. In one extreme, the nodes may be entirely disjoint (as illustrated in Figure 1(b)), in which case we would want to prefetch every list node. Another possibility might be that each list shares a long common "tail" starting with the second list node (as illustrated in Figure 1(c)). In this latter case, there would be significant temporal locality (assuming the cache is large enough to contain the common tail), and ideally we would only want to prefetch the nodes in the common tail during the first list traversal (i.e. when `i=1`). Unfortunately, despite the significant progress that has been made recently in pointer analysis techniques for heap-allocated objects [6, 8, 10], compilers are still not sophisticated enough to differentiate these two cases automatically. In general, analyzing the addresses of heap-allocated objects is a very difficult problem for the compiler.

(a) Traversing an RDS



(b) Prefetching 3 Nodes Ahead

```
while (p) {
  prefetch(p→next→next→next);
  work(p→data);
  p = p→next;
}
```

(c) Prefetching 1 Node Ahead

```
while (p) {
  prefetch(p→next);
  work(p→data);
  p = p→next;
}
```

Figure 2: Illustration of the pointer-chasing problem.

2.1.2 Scheduling

Our ability to schedule prefetches for an RDS is also constrained by the fact that nodes are linked together through pointers. For example, consider the case shown in Figure 2(a), where assuming that three nodes worth of computation is needed to hide the latency, we would like to initiate a prefetch for node n_{i+3} while we are visiting node n_i. The problem is that to compute the address of node n_{i+3}, we must first dereference a pointer in node n_{i+2}, and to do that, we must first dereference a pointer in node n_{i+1}, etc. As a result, one cannot prefetch (or fetch) a future node until all nodes between it and the current node have been fetched. However, the very act of touching these intermediate nodes means that we cannot tolerate the latency of fetching more than one node ahead. For example, the prefetching code shown in Figure 2(b) will not hide any more latency than the code in Figure 2(c).[1] In fact, the code in Figure 2(c) is likely to run faster since it has less instruction overhead. This example illustrates what we refer to as the *pointer-chasing problem.*

Since scheduling RDS prefetches is such a difficult problem, we make it the primary focus of this paper. Improvements in analysis tend to reduce prefetching overhead by eliminating unnecessary prefetches. However, without sufficient scheduling techniques, there will be no upside to prefetching and hence reducing overhead will be irrelevant. Fortunately, as we discuss in the next subsection, there are techniques for scheduling prefetches that avoid the pointer-chasing problem.

2.2 Overcoming the Pointer-Chasing Problem

Let us formalize the pointer-chasing problem as follows. At a given RDS node n_i with address A_i, we wish to prefetch the node n_{i+d} that will be visited d nodes after n_i. We choose d (the *prefetching distance*) to be just large enough to hide the cache miss latency: $d = \lceil \frac{L}{W} \rceil$, where L is the expected miss latency and W is the estimated amount of computation between node accesses in cycles. To prefetch n_{i+d}, we must compute its address A_{i+d} based on the information available at n_i. The relationship between A_i and A_{i+d} can be expressed as:

$$A_{i+d} = \mathcal{F}(d, A_i)$$

where $\mathcal{F}$ is an address generating function.

A key factor in whether prefetch scheduling is effective is the *number of pointer-chain dereferences* required within the RDS to evaluate $\mathcal{F}(d, A_i)$, which we denote as $\|\mathcal{F}\|$. To overcome the pointer-chasing problem, we would like $\|\mathcal{F}\|$ to be as small as possible. If $\mathcal{F}$ is implemented by following the pointer chain from n_i to n_{i+d}, then $\|\mathcal{F}\| = d$. Instead, we will consider the cases where $\|\mathcal{F}\| = 1$ and $\|\mathcal{F}\| = 0$ (other values of $\|\mathcal{F}\|$ are possible, but do not appear to be interesting).

The case where $\|\mathcal{F}\| = 1$ means that only one pointer dereference is needed within the RDS to compute A_{i+d} at node n_i. This implies that n_i needs a direct pointer to n_{i+d}—we call this pointer a *jump-pointer.* Jump-pointers can occur either *naturally* or *artificially* with respect to the RDS: a natural jump-pointer is a pointer that already exists in n_i, whereas an artificial jump-pointer is added to n_i for the purpose of prefetching. With natural jump-pointers, we are using one of the pointers at n_i to *approximate* A_{i+d}. The advantage is that no extra storage or computation is needed to create a natural jump-pointer, but unfortunately the effectiveness of prefetching may be limited by the accuracy of this approximation. In contrast, we require additional storage and computation to add artificial jump-pointers to an RDS, but hopefully these pointers will yield A_{i+d} more precisely (particularly if the structure of the RDS does not change rapidly between times when the artificial jump-pointers are set).

The case where $\|\mathcal{F}\| = 0$ means that *no* pointer dereferences are required to compute A_{i+d} at n_i. This is obviously a good case, but how can one compute the address of a heap-allocated object (which normally can reside at an arbitrary address) without dereferencing any pointers? The answer is that we must have special knowledge of an RDS's layout in memory such that A_{i+d} can be directly implied from A_i and d.[2] There are many ways to accomplish this. For example, one could map a tree into an array structure such that there was a one-to-one mapping between the tree position and an array index. We will discuss the details of the approach we take later in Section 2.2.3.

In the remainder of this section, we propose three prefetching schemes with various $\|\mathcal{F}\|$ which avoid the pointer-chasing problem: *greedy prefetching* corresponds to $\|\mathcal{F}\| = 1$ using natural jump-pointers; *history-pointer prefetching* corresponds to $\|\mathcal{F}\| = 1$ using artificial jump-pointers; and *data-linearization prefetching* is a case where $\|\mathcal{F}\| = 0$.

2.2.1 Greedy Prefetching

In a k-ary RDS, each node contains k pointers to other nodes. Greedy prefetching exploits the fact that when $k > 1$, only one of these k pointers can be immediately followed by control flow as the next node in the traversal. Hence the remaining $k-1$ pointers serve as natural jump-pointers, and can be prefetched immediately upon first visiting a node. Although none of these jump-pointers may actually point to n_{i+d}, hopefully each of them points to $n_{i+d'}$ for some $d' > 0$. If $d' < d$, then the latency may be partially hidden; if $d' \geq d$, then we expect the latency to be fully hidden, provided that the node is not displaced from the cache before it is referenced (which may occur if $d' \gg d$).

To illustrate how greedy prefetching works, consider the preorder traversal of a binary tree (i.e. $k = 2$), where Figure 3(a) shows the code with greedy prefetching added. Assuming that the computation in `process()` takes half as long as the cache miss latency, we would want to prefetch two nodes ahead (i.e. $d = 2$) to fully hide the latency. Figure 3(b) shows the caching behavior of each node. We obviously suffer a full cache miss at the root node (node 1), since there was no opportunity to fetch it ahead of time. However, we would only suffer half of the miss penalty ($\frac{L}{2}$) when we visit node 2, and no miss penalty when we eventually visit node 3 (since the time to visit the subtree

[1] Assuming that nodes are not larger than cache lines; if they are, then prefetching further ahead can potentially result in a pipelining benefit.

[2] We may also need to take other information into account, such as the traversal order, but nothing can involve dereferencing a pointer within n_i.

(a) Code with Greedy Prefetching

```
preorder(treeNode * t) {
  if (t != NULL) {
    prefetch(t→left);
    prefetch(t→right);
    process(t→data);
    preorder(t→left);
    preorder(t→right);
  }
}
```

(b) Cache Miss Behavior

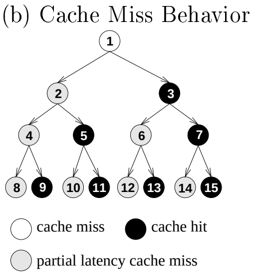


Figure 3: Illustration of greedy prefetching.

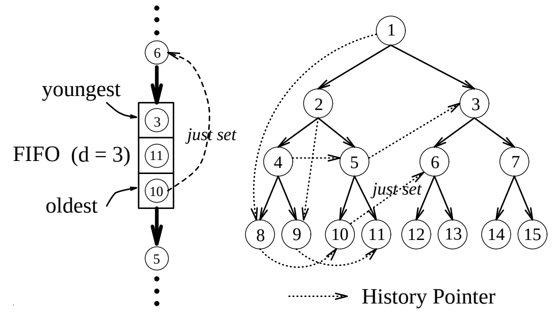


Figure 4: Example showing the update of history-pointers

rooted at node 2 is greater than L). In this example, the latency is fully hidden for roughly half of the nodes, and reduced by 50% for the other half (minus the root node). If we generalize this example to a k-ary tree, we would expect the fraction of nodes where latency is fully hidden to be roughly $\frac{k-1}{k}$ (assuming that prefetched nodes are generally not displaced from the cache before they are referenced). Hence a larger value of k is likely to improve the performance of greedy prefetching, since more natural jump-pointers are available.

Greedy prefetching offers the following advantages: (i) it has low runtime overhead, since no additional storage or computation is needed to construct the natural jump-pointers; (ii) it is applicable to a wide variety of RDSs, regardless of how they are accessed or whether their structure is modified frequently; and (iii) it is relatively straightforward to implement in a compiler (in fact, we have implemented it in the SUIF compiler, as we describe later in Section 3). The main disadvantage of greedy prefetching is that it does not offer precise control over the prefetching distance, which is the motivation for our next algorithm.

2.2.2 History-Pointer Prefetching

Rather than relying on natural jump-pointers to approximate A_{i+d}, we can potentially synthesize more accurate jump-pointers based on the actual RDS traversal patterns, while still achieving $\|\mathcal{F}\| = 1$. The idea behind the *history-pointer prefetching* scheme is that we create a new jump-pointer (called a *history-pointer*) in n_i which contains the observed value of A_{i+d} during a recent traversal of the RDS. (Note that we could potentially store multiple artificial jump-pointers in n_i to account for multiple traversal orderings.) On subsequent traversals of the RDS, we prefetch the nodes pointed to by these history-pointers. This scheme is most effective when the traversal pattern does not change rapidly over time, in which case the history-pointer in n_i is likely to point to either n_{i+d} or else hopefully a node that will be visited soon. On the other hand, if the structure of the RDS changes radically between traversals, the history-pointers might not be effective.

To construct the history-pointers, we maintain a FIFO queue of length d which contains pointers to the last d nodes that have just been visited. When we visit a new node n_i, the oldest node in the queue will be n_{i-d} (i.e. the node visited d nodes earlier), and hence we update the history-pointer of n_{i-d} to point to n_i. After the first complete traversal of the RDS, all of the history-pointers will be set. Figure 4 illustrates a snapshot of this bookkeeping process for the tree shown earlier in Figure 3. Assuming that $d = 3$ and that we have just reached node 6, we would now update the history-pointer of the oldest node in the 3-entry queue (node 10) to point to node 6.

Comparing the performance of this scheme with greedy prefetching, history-pointer prefetching offers no improvement on the first traversal of an RDS, since the history-pointers have yet to be set (greedy prefetching would hide some fraction of the latency).[3] However, on subsequent traversals of the RDS, history-pointer prefetching will hide nearly all of the latency, whereas greedy prefetching will continue to hide only a fraction of the latency.

While history-pointer prefetching offers the potential advantage of improved latency tolerance, this comes at the expense of two additional forms of overhead: (i) execution overhead to construct the history-pointers, and (ii) space overhead for storing these new pointers. To minimize execution overhead, we can potentially update the history-pointers less frequently, depending on how rapidly the structure of the RDS changes. In one extreme, if the RDS never changes, we only need to set the history-pointers once. The problem with space overhead is that it potentially worsens the caching behavior. The desire to eliminate this space overhead altogether is the motivation for our next prefetching scheme.

2.2.3 Data-Linearization Prefetching

The goal of *data-linearization prefetching* is to achieve an $\mathcal{F}$ such that A_{i+d} can be predicted precisely, but without requiring any pointer dereferences (i.e. $\|\mathcal{F}\| = 0$). Another advantage of this scheme is that it improves spatial locality. The basic idea is to map heap-allocated nodes that are likely to be accessed close together in time into contiguous memory locations. With this mapping, one can easily predict A_{i+d} and hence prefetch it early enough.

Recall that the address of an array element $x[i + d]$ can be computed relative to $x[i]$ as follows (using C-like syntax):

$$\&x[i + d] = \&x[i] + d \times sizeof(x[0]) \qquad (1)$$

Therefore, if we can map the RDS onto an array x of nodes such that $A_i = \&x[i]$, no pointer dereference is needed to compute A_{i+d}—we simply need two arithmetic operations per prefetch address.

The question is how and when can this mapping (which we call *data linearization*) be performed? In theory, one could dynamically remap the data even after the RDS has been initially constructed, but doing so may result in large runtime overheads and may also violate program semantics.[4] Instead, the best time to map the nodes is at creation time, which is appropriate if either the creation order already matches the traversal order, or if it can be safely reordered to do so. Since dynamic remapping is expensive (or impossible), this scheme obviously works best if the structure of the RDS changes only slowly (or not at all).

[3] Hence we may want to use greedy prefetching for the first traversal of an RDS when the history-pointers are being initialized.

[4] All pointers to these objects would also need to be updated, and understanding pointer aliasing for heap-allocated objects is quite difficult for the compiler.

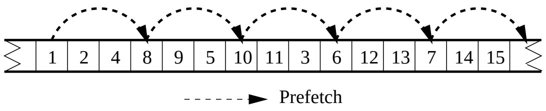


Figure 5: Illustration of data-linearization prefetching

If the RDS does change radically, the program will still behave correctly, but prefetching will not improve performance.

Figure 5 illustrates how data-linearization prefetching works for the tree shown earlier in Figures 3 and 4. The ordering of nodes in the array corresponds to the pre-order traversal order. To prefetch d nodes ahead, one simply uses equation (1) to compute A_{i+d}. In addition, if a single cache line can hold $m > 1$ nodes, we can exploit this spatial locality by only issuing a prefetch once every m nodes. If we are traversing the RDS inside a loop, we can accomplish this by unrolling the loop by a factor of m (similar to what is done in array-based prefetching [15]). For a traversal through recursion, one could potentially keep track of the number of nodes visited between prefetches, but the overhead of doing so may be comparable to simply issuing a prefetch for every node. The arrows in Figure 5 indicate the desired prefetches when $d = 3$ and $m = 3$.

2.3 Summary

The nature of recursive data structures makes both the analysis and scheduling of prefetches quite challenging. Before attempting to minimize prefetching overhead through improved analysis, we must first maximize the latency-hiding gain through effective prefetch scheduling techniques. The fundamental problem in scheduling prefetches for RDSs is the *pointer-chasing problem*, which we formalize as the number of pointer-chain dereferences required to compute a prefetch address ($\|\mathcal{F}\|$). Based on our desire to minimize $\|\mathcal{F}\|$, we have identified three promising prefetching schemes: *greedy prefetching* ($\|\mathcal{F}\| = 1$ with natural jump-pointers), *history-pointer prefetching* ($\|\mathcal{F}\| = 1$ with artificial jump-pointers), and *data-linearization prefetching* ($\|\mathcal{F}\| = 0$).

Of these three schemes, greedy prefetching is perhaps the most widely applicable since it does not rely on traversal history information, and it requires no additional storage or computation to construct prefetch addresses. For these reasons, we have implemented a version of greedy prefetching as an automatic compiler pass, and we will simulate the other two algorithms by hand to compare their performance with greedy prefetching. In the next section, we describe the implementation details of our greedy prefetching compiler pass.

3 Implementation of Greedy Prefetching

Our implementation of greedy prefetching within the SUIF compiler [20] consists of an *analysis* phase to recognize RDS accesses, and a *scheduling* phase to insert prefetches.

3.1 Analysis: Recognizing RDS Accesses

To recognize RDS accesses, the compiler uses both *type declaration* information to recognize which data objects are RDSs, and *control structure* information to recognize when these objects are being traversed. An RDS type is a record type r containing at least one pointer that points either directly or indirectly to a record type s. (Note that r and s are not restricted to be the same type, since RDSs may be comprised of heterogeneous nodes.) For example, the type declarations in Figure 6(a) and Figure 6(b) would be recognized as RDS types, whereas Figure 6(c) would not.[5]

```
struct T {
  int data;
  struct T *left;
  struct T *right;
}
```
(a) RDS type

```
struct A {
  int i;
  struct B **kids[8];
}
```
(b) RDS type

```
struct C {
  int j;
  double f;
}
```
(c) Not RDS type

Figure 6: Examples of whether type declarations are recognized as being RDS types.

After discovering data structures with the appropriate types, the compiler then looks for control structures that are used to traverse the RDSs. In particular, the compiler looks for *loops* or *recursive procedure calls* such that during each new loop iteration or procedure invocation, a pointer p to an RDS is assigned a value resulting from a dereference of p—we refer to this as a *recurrent pointer update*. This heuristic corresponds to how RDS codes are typically written. To detect recurrent pointer updates, the compiler propagates pointer values using a simplified (but less precise) version of earlier pointer analysis algorithms [7, 12].

(a)
```
while (1) {
  listNode *m;
  ...
  m = l→next;
  l = m→next;
  ...
}
```
(b)
```
for (...) {
  listNode *n;
  ...
  n = g(n);
  ...
}
```
(c)
```
f(treeNode *t) {
  ...
  f(t→left);
  f(t→right);
  ...
}
```
(d)
```
k(treeNode tn) {
  ...
  k(*(tn.left));
  k(*(tn.right));
  ...
}
```

Figure 7: Examples of recognizable control structures for RDS traversals.

Figure 7 shows some example program fragments that our compiler treats as RDS accesses. In Figure 7(a), `l` is updated to `l→next→next` inside the while-loop. In Figure 7(b), `n` is assigned the result of the function call `g(n)` inside the for-loop. (Since our implementation does not perform interprocedural analysis, it assumes that `g(n)` results in a value `n→...→next`.) In Figure 7(c), two dereferences of the function argument `t` are passed as the parameters to two recursive calls. Figure 7(d) is similar to Figure 7(c), except that a record (rather than a pointer) is passed as the function argument.

Ideally, the next step would be to analyze data locality across RDS nodes—e.g., to distinguish the two cases shown in Figure 1—to eliminate any unnecessary prefetches. Although we have not automated this step in our compiler, we will evaluate its potential benefit later in Section 5.3.

3.2 Scheduling Prefetches

Once RDS accesses have been recognized, the compiler inserts greedy prefetches as follows. At the point where an RDS object

[5]The compiler may fail to recognize cases with explicit type casting—e.g., casting `j` to be of type `(struct C*)` in Figure 6(c)—but such cases do not appear to be common.

Table 1: Benchmark characteristics.

Benchmark	Description	Recursive Data Structures Used	Input Data Set	Node Memory Allocated
BH	Barnes-Hut's N-body force-calculation algorithm	Heterogenous octree	4K bodies	4128 x 136 B = 548 KB + 2021 x 88 B = 173 KB
Bisort	Sorts two disjoint bitonic sequences and then merges them	Binary tree	250,000 integers	131,017 x 12 B = 1535 KB
EM3D	Simulates the propagation of electromagnetic waves in a 3D object	Singly-linked lists	2000 H-nodes, 100 E-nodes, 75% local	4000 x 28 B = 109 KB + 400,000 x 4 B = 1562 KB
Health	Simulation of the Columbian health care system	Four-way tree and doubly-linked lists	max. level = 5, max. time = 500	341 x 100 B = 33 KB + 57,111 x 16 B = 892 KB
MST	Finds the minimum spanning tree of a graph	Array of singly-linked lists	512 nodes	512 x 20 B = 10 KB
Perimeter	Computes perimeters of regions in images	A quadtree	4K x 4K image	235,717 x 28 B = 6445 KB
Power	Solves the power system optimization problem	Multi-way tree and singly-linked lists	10,000 customers	200 x 56 B = 11 KB + 1000 x 96 B = 94 KB + 10,000 x 32 B = 313 KB
TreeAdd	Sums the values distributed on a tree	Binary tree	1024K nodes	1,048,576 x 12 B = 12,288 KB
TSP	Traveling salesman problem	Binary tree and doubly-linked lists	100,000 cities	131,071 x 40 B = 5120 KB
Voronoi	Computes the voronoi diagram of a set of points	Binary tree	20,000 points	633,032 x 16 B = 9891 KB + 32,768 x 32 B = 1024 KB

```
while (1) {                    ⟹      while (1) {
  work(l→data);                          prefetch(l→next);
  l = l→next;                            work(l→data);
}                                        l = l→next;
                                       }
```

(a) Loop

```
f(treeNode *t) {               ⟹      f(treeNode *t) {
  treeNode *q;                           treeNode *q;
  if (test(t→data))                      prefetch(t→left);
    q = t→left;                          prefetch(t→right);
  else q = t→right;                      if (test(t→data))
  if (q != NULL)                           q = t→left;
    f(q);                                else q = t→right;
}                                        if (q != NULL)
                                           f(q);
                                       }
```

(b) Procedure

Figure 8: Examples of greedy prefetch scheduling.

is being traversed (i.e. where the recurrent pointer update occurs), the compiler inserts prefetches of all pointers within this object that point to RDS-type objects (these are the natural jump-pointers[6]) at the earliest points where these addresses are available within the surrounding loop or procedure body. The availability of prefetch addresses is computed by propagating the earliest generation points of pointer values along with the values themselves. Two examples of greedy prefetch scheduling are shown in Figure 8.

4 Experimental Framework

To evaluate the performance of our three prefetching schemes, we performed detailed cycle-by-cycle simulations of the entire Olden benchmark suite [17] on a dynamically-scheduled, superscalar processor similar to the MIPS R10000. The Olden benchmark suite contains ten pointer-based applications written in C, which are briefly summarized in Table 1. The rightmost column in Table 1 shows the number and size of each node type that was dynamically allocated.

[6]Note that we do not prefetch *all* pointers within an RDS object—only the ones that point to other RDS nodes (potentially of different types than the given object).

Table 2: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 2 Memory, 1 Branch
Reorder Buffer Size	32
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	2-bit Counters

Memory Parameters	
Primary Instr and Data Caches	16KB, 2-way set-associative
Unified Secondary Cache	512KB, 2-way set-associative
Line Size	32B
Primary-to-Secondary Miss	12 cycles
Primary-to-Memory Miss	75 cycles
Data Cache Miss Handlers	8
Data Cache Banks	2
Data Cache Fill Time (Requires Exclusive Access)	4 cycles
Main Memory Bandwidth	1 access per 20 cycles

Our simulation model varies slightly from the actual MIPS R10000 (e.g., we model two memory units, and we assume that all functional units are fully-pipelined), but we do model the rich details of the processor including the pipeline, register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, the memory hierarchy (including contention), etc. The parameters of our model are shown in Table 2. We use *pixie* [18] to instrument the optimized MIPS object files produced by the compiler, and pipe the resulting trace into our simulator.

To minimize the impact of store stalls during the initialization of dynamically-allocated objects, we use our own memory allocator for these experiments which is similar to `mallopt` provided in the Irix C library [19], but also contains built-in prefetching to avoid such store misses. This optimization alone led to dramatic improvements (greater than two-fold speedups) over using `malloc` for the majority of the applications—particularly the ones that frequently allocate small objects.

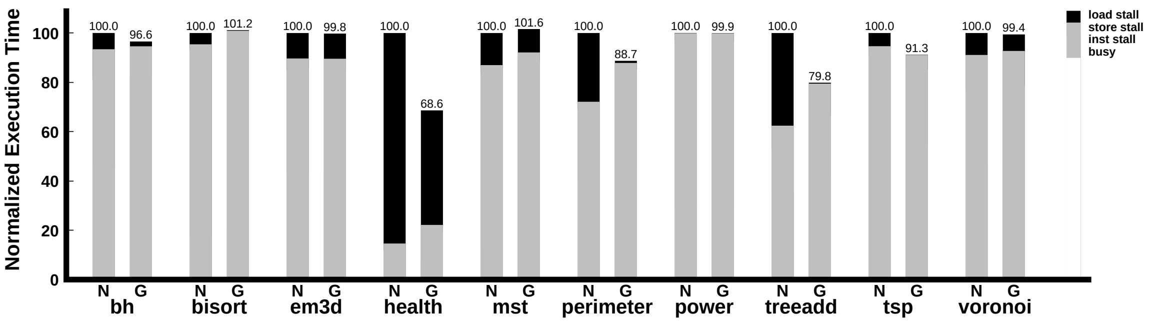
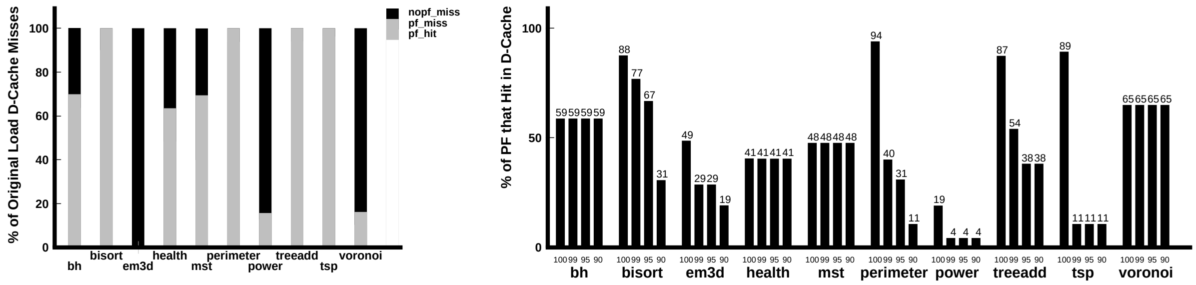


Figure 9: Performance of compiler-inserted greedy prefetching (**N** = no prefetching, **G** = greedy prefetching).



(a) Coverage Factor

(b) Unnecessary Prefetches

Figure 10: Additional performance metrics for evaluating greedy prefetching.

5 Experimental Results

We now present results from our simulation studies. We start by evaluating the performance of compiler-inserted greedy prefetching, and then compare this with hand-inserted versions of history-pointer prefetching and data-linearization prefetching. Next, we evaluate the potential performance gains from better analysis to reduce unnecessary prefetches. Finally, we explore the performance impact of architectural support.

5.1 Performance of Compiler-Inserted Greedy Prefetching

The results of our first set of experiments are shown in Figures 9 and 10. Figure 9 shows the overall performance improvement offered by greedy prefetching, where the two bars correspond to the cases without prefetching (**N**) and with greedy prefetching (**G**). These bars represent execution time normalized to the case without prefetching, and they are broken down into four categories explaining what happened during all potential graduation slots.[7] The bottom section (*busy*) is the number of slots when instructions actually graduate, the top two sections are any non-graduating slots that are immediately caused by the oldest instruction suffering either a load or store miss,[8] and the *inst stall* section is all other slots where instructions do not graduate. Note that the *load stall* and *store stall* sections are only a first-order approximation of the performance loss due to cache stalls, since these delays also exacerbate subsequent data dependence stalls.

As we see in Figure 9, half of the applications enjoy a speedup ranging from 4% to 45% (the other half are within 2% of their original performance). For the applications with the largest memory stall penalties (i.e. `health`, `perimeter`, and `treeadd`), much of this stall time has been eliminated. In the cases of `bisort` and `mst`, prefetching overhead more than offset the reduction in memory stalls (thus resulting in a slight performance degradation), but this was not a problem in the other eight applications. (Later, in Section 5.3, we will explore how to further reduce this overhead.)

To understand the performance results in greater depth, Figure 10 presents two additional performance metrics. Figure 10(a) breaks down the original primary cache misses into three categories: (i) those that are prefetched and subsequently hit in the primary cache (*pf_hit*), (ii) those that are prefetched but remain primary misses (*pf_miss*), and (iii) those that are not prefetched (*nopf_miss*). The sum of the *pf_hit* and *pf_miss* cases is also known as the *coverage factor*, which ideally should be 100%. For `em3d`, `power`, and `voronoi`, the coverage factor is quite low (under 20%) because most of their misses are caused by array or scalar references—hence prefetching RDSs yields little improvement. In all other cases, the coverage factor is above 60%, and in four cases we achieve nearly perfect coverage. If the *pf_miss* category is large, this indicates that prefetches were not scheduled effectively—either they were issued *too late* to hide the latency, or else they were *too early* and the prefetched data was displaced from the cache before it could be referenced. This category is most prominent in `mst`, where the compiler is unable to prefetch early enough during the traversal of very short linked lists within a hash table. Since the natural jump-pointers in greedy prefetching offer little control over prefetching distance, it is not surprising that scheduling is imperfect—in fact, it is encouraging that the *pf_miss* fractions are this low. Later, in Section 5.2, we will

[7] The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles. We focus on graduation rather than issue slots to avoid counting speculative operations that are squashed.

[8] Store misses only stall the processor when the 32-entry memory issue buffer is full.

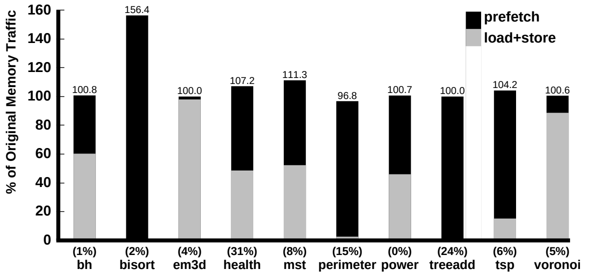


Figure 11: Increase in total memory traffic due to greedy prefetching. Numbers below the bars indicate the memory utilization with greedy prefetching.

explore techniques for improving prefetch scheduling.

To help evaluate the costs of prefetching, Figure 10(b) shows the fraction of dynamic prefetches that are *unnecessary* because the data is found in the primary cache. For each application, we show four different bars indicating the total (dynamic) unnecessary prefetches caused by static prefetch instructions with hit rates up to a given threshold. Hence the bar labeled "**100**" corresponds to all unnecessary prefetches, whereas the bar labeled "**99**" shows the total unnecessary prefetches if we exclude prefetch instructions with hit rates over 99%, etc. This breakdown indicates the potential for reducing overhead by eliminating static prefetch instructions that are clearly of little value. For example, eliminating prefetches with hit rates over 99% would eliminate over half of the unnecessary prefetches in `perimeter`, thus decreasing overhead significantly. In contrast, reducing overhead with a flat distribution (e.g., `bh`) is more difficult since prefetches that sometimes hit also miss at least 10% of the time (therefore, eliminating them may sacrifice some latency-hiding benefit). We will quantify the benefit of eliminating unnecessary prefetches later in Section 5.3.

To further evaluate the costs of greedy prefetching, Figure 11 shows its impact on memory bandwidth. Ideally, prefetching will not increase memory traffic, since the original memory references will simply be converted into prefetches. (In fact, previous studies have demonstrated that prefetching can actually *reduce* the memory traffic in a shared-memory multiprocessor through exclusive-mode hints [15].) However, since the natural jump-pointers used by greedy prefetching may point to nodes that will not be accessed in the near future (or perhaps not at all), greedy prefetching can potentially increase the memory bandwidth demands through useless prefetches. As we see in Figure 11, greedy prefetching has increased memory traffic by less than 12% for all but one application (in one case—`perimeter`—the traffic actually decreased slightly due to fortuitous cache replacement behavior in the set-associative caches). In the case of `bisort`, where we do see a noticeable increase of 56%, the total memory utilization still remains so low with greedy prefetching (2%) that there is no impact on performance. Hence greedy prefetching does not appear to be suffering from memory bandwidth problems.

Although space constraints prevent a detailed discussion of each application, we briefly summarize some of the highlights (code fragments are shown in Figure 12).

bh: Nodes of an octree are traversed in `bh_walk()`, and 70% of load stalls occur in `bh_test()` and `bh_work()` (see Figure 12(a)). The compiler immediately prefetches all eight children of the current node `t` before `bh_test()` is called. Although 59% of prefetches are unnecessary, the overhead remains low and there is a 4% speedup.

(a) **bh**

```
bh_walk(..., node*t, ...) {
    prefetch(t→children[0]);
    prefetch(t→children[1]);
    prefetch(t→children[2]);
    prefetch(t→children[3]);
    prefetch(t→children[4]);
    prefetch(t→children[5]);
    prefetch(t→children[6]);
    prefetch(t→children[7]);
    if (bh_test(p)) {
        for (k=0; k<8; k++){
            r = t→children[k];
            if (r)
                bh_walk(..., r, ...);
        }
    } else
            bh_work(..., t, ...);
}
```

(b) **bisort**

```
int Bimerge(root,spr_val,dir)
    rv = root→value;
    pl = root→left; pr = root→right;
    ...
    while ((pl != NIL)) {
        prefetch(pl→left);
        prefetch(pl→right));
        prefetch(pr→left);
        prefetch(pr→right);
        lv = pl→value;
        pll = pl→left; plr = pl→right;
        rv = pr→value;
        prl = pr→left; prr = pr→right;
        if (...) {
            if (...) {
                SwapValRight(pl,pr,plr,prr,lv,rv);
                pl = pll; pr = prl;
            } else {
                pl = plr; pr = prr;
            }
        } else {
            if (...) {
                SwapValLeft(pl,pr,pll,prl,lv,rv);
                pl = plr; pr = prr;
            } else {
                pl = pll; pr = prl;
            }
        }
    }
    if ((root→left != NIL)) {
        prefetch(root→left);
        prefetch(root→right);
        ...
        rl = root→left; rr = root→right;
        ...
        root→value=Bimerge(rl,value,dir);
        spr_val=Bimerge(rr,spr_val,dir);
    }
    ...
}
```

(c) **health**

```
void waiting(Village *village, List *list)
    while (list != NULL) {
        prefetch(list→forward);
        i = village→hosp.free_personnel;
        p = list→patient;
        if (i > 0) {
            t = village→hosp.free_personnel;
            village→hosp.free_personnel = t-1;
            p→time_left = 3;
            p→time = t + 3;
            l = &(village→hosp.waiting);
            removeList(l, p);
            l = &(village→hosp.assess);
            addList(l, p);
        } else {
            t = p→time;
            p→time = t + 1;
        }
        list = list→forward;
    }
}
```

(d) **mst**

```
void *HashLookup(int key, Hash hash) {
    j = (hash→mapfunc)(key);
    for (ent = hash→array[j];
        ent && ent→key!=key;
        ent=ent→next)
            prefetch(ent→next);
    if (ent) return ent→entry;
    return NULL;
}
```

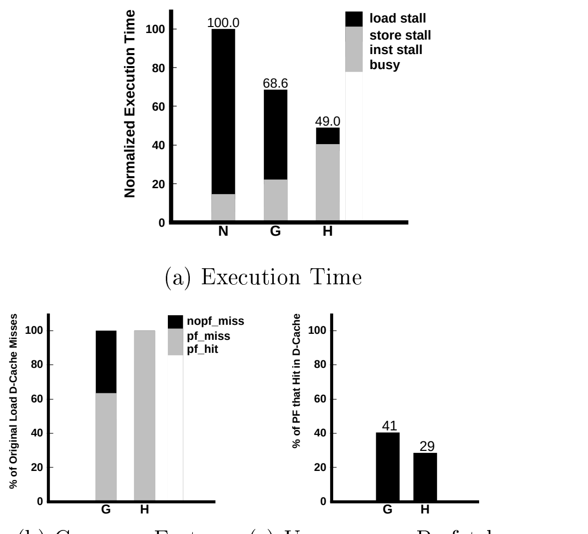
Figure 12: Abstract representation of the output of the greedy prefetching compiler for some interesting code fragments in the Olden benchmarks.

bisort: The main RDS is a binary tree, and the important cache misses occur in `Bimerge()`, which contains both loops and recursion (see Figure 12(b)). The four "grandchildren" of `root` are prefetched early in the while loop. Although load misses are completely hidden, execution time increases by 1.2% due to unnecessary prefetching overhead. Locality analysis might help this case by recognizing that a portion of data accessed in the recursive calls has already been brought into the cache by the while loop.

health: Over 90% of load stalls are due to linked-list accesses inside `waiting()` (see Figure 12(c)). Despite a noticeable increase in overhead, the 50% reduction in load stalls results in a large speedup.

mst: 90% of load stalls occur in `HashLookup()`, where it searches for an item in an array of linked lists (see Figure 12(d)). Although the compiler prefetches `ent→next`, only a small portion of the latency can be hidden since the loop body is so small. This appears to be a general problem with hash tables, and prefetching prior to the hash function invocation is beyond the scope of our algorithm.

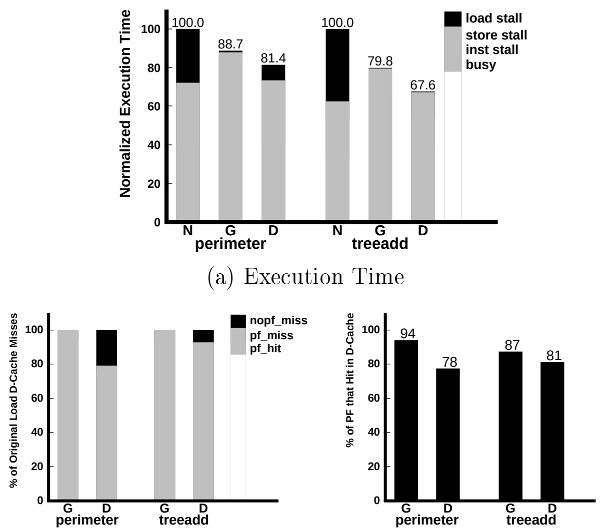
perimeter: A quadtree is traversed through recursive procedure calls. All load misses are covered, but the 94% unnecessary prefetches result in significant overheads. There are two reasons for the unnecessary prefetches: (i) the same parts of the quadtree can be visited through different recursive procedures, thus resulting in unanticipated data locality; and (ii) each node contains a pointer to its parent, which the compiler prefetches along with the four child pointers, but the parent is already in the cache.



(a) Execution Time

(b) Coverage Factor (c) Unnecessary Prefetches

Figure 13: Performance of `health` with history-pointer prefetching (**N** = no prefetching, **G** = greedy prefetching, **H** = history-pointer prefetching).



(a) Execution Time

(b) Coverage Factor (c) Unnecessary Prefetches

Figure 14: Performance of `perimeter` and `treeadd` with data-linearization prefetching (**N** = no prefetching, **G** = greedy prefetching, **D** = data-linearization prefetching).

`tsp`: Each RDS node contains four pointers: two for binary tree-like accesses, and two for doubly-linked list-like accesses. The abundant unnecessary prefetches occur for the same reasons as `perimeter`. Prefetching reduces the *inst stall* time (see Figure 9) in this case by accelerating data dependency chains.

In summary, we have seen that automatic compiler-inserted prefetching can result in significant speedups for applications containing recursive data structures. In the next two sections, we will evaluate techniques for increasing these gains even further.

5.2 History-Pointer Prefetching and Data-Linearization Prefetching

To quantify the performance potential of the more sophisticated prefetching schemes proposed earlier in Section 2.2, we applied them by hand to our applications. Figure 13 shows the performance of the one application that improves under history-pointer prefetching: `health`. History-pointer prefetching works in this case because the structure of the lists accessed in `waiting()` (see Figure 12(c)) remains unchanged throughout the over ten thousand times it is called. Two history-pointers are added to the `List` record: one for prefetching `list` and one for prefetching `list`→`patient`, with prefetching distances of four and two, respectively. As we see in Figure 13, history-pointer prefetching results in a 40% speedup over greedy prefetching through better miss coverage and fewer unnecessary prefetches. Coverage is improved because `list`→`patient` is successfully prefetched—under greedy prefetching, the compiler does not recognize `list`→`patient` as an RDS access,[9] and even if it did, there would not be sufficient time to hide the latency. Although history-pointer prefetching has a smaller fraction of unnecessary prefetches, it has more overhead than greedy prefetching due to the extra work required to maintain the history-pointers.

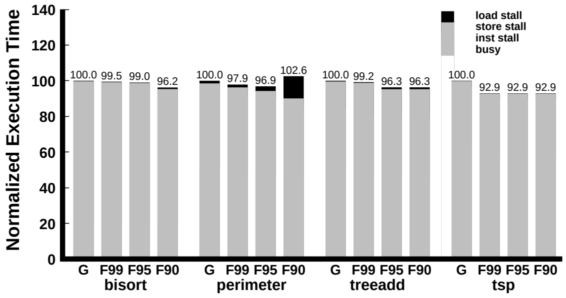
[9] The reason why greedy prefetching does not recognize `list`→`patient` as an RDS access is that there is no recurrent pointer update for the `patient` object type. As we discussed earlier in Section 3, the compiler does not prefetch pointers unless they point to RDSs.

Data-linearization prefetching is applicable to both `perimeter` and `treeadd`, because the creation order is identical to the major subsequent traversal order in both cases. As a result, data linearization does not require changing the data layout in these cases (hence spatial locality is unaffected). As we see in Figure 14, data-linearization prefetching offers additional speedups ranging from 9% to 18% through fewer unnecessary prefetches (and hence less prefetching overhead), while still maintaining good coverage factors. Unnecessary prefetches are reduced because only one prefetch is issued per node (whereas greedy prefetching may follow multiple natural jump-pointers), and coverage suffers slightly because there are multiple traversal orders, and data linearization only captures the most common one. Overall, we see that both history-pointer prefetching and data-linearization can potentially offer significant improvements over greedy prefetching when applicable.

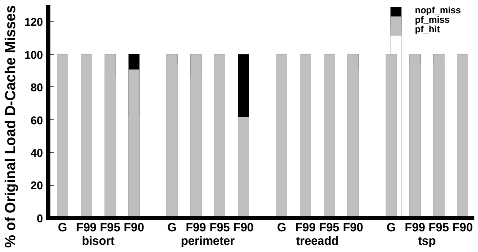
5.3 Reducing Overhead Through Locality Analysis

Our compiler currently does not attempt to analyze data locality across RDS node accesses. As a result, we may prefetch nodes unnecessarily that already reside in the cache (as discussed earlier in Section 5.1). For numeric applications, sophisticated locality analysis techniques have been combined with loop splitting techniques to isolate the dynamic iterations that should be prefetched [16]. Unfortunately, the control structures in RDS codes are less amenable to isolating dynamic node visitations, so our only option may be to eliminate static prefetch instructions altogether. This makes sense for prefetches that are almost always unnecessary (i.e. have very high hit rates).

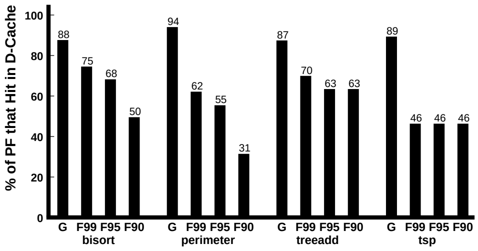
To estimate the performance potential of exploiting locality information, we used memory feedback information from our simulator to eliminate prefetch instructions with hit rates above a certain threshold from the greedy prefetching code. Figure 15 shows our results for the four applications that were affected by setting this threshold to 99%, 95%, and 90% hit rates. As we see in Figure 15, eliminating prefetches with hit rates above



(a) Execution Time (Normalized to **G**)



(b) Coverage Factor



(c) Unnecessary Prefetches

Figure 15: Performance of feedback-guided greedy prefetching on four benchmarks (**G** = greedy prefetching, **Fxx** = greedy prefetching where static prefetch instructions with hit rates over xx% have been eliminated).

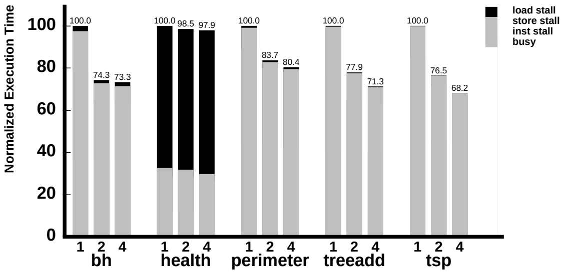
95% improves performance by 1-7% for these applications by eliminating unnecessary prefetches without sacrificing much coverage. However, eliminating prefetches with hit rates over 90% does hurt performance in `perimeter`, since the coverage factor drops dramatically. Therefore, improved locality analysis may help performance by eliminating prefetches that are almost always unnecessary (e.g., the "`parent`" pointer in `perimeter`), but without more powerful techniques for isolating dynamic node visitations, the gains do not appear to be as large as with numeric codes.

5.4 Architectural Support for Prefetching RDSs

We now explore the impact of two key architectural issues on the performance of our greedy prefetching algorithm.

5.4.1 Number of Memory Functional Units

Prefetching RDSs increases the demand for memory functional units in two ways. In addition to the prefetches themselves, we may also need additional loads (e.g., for jump-pointers) to compute the prefetch addresses. Figure 16(b) shows that an average of 0.4-1.4 loads were required per prefetch. Although the actual



(a) Execution Time (Normalized to case **1**)

Application	Insts per Prefetch	Loads per Prefetch
bh	3.6	0.5
bisort	2.9	1.3
em3d	4.8	1.4
health	3.6	0.5
mst	5.9	0.7
perimeter	2.1	0.5
power	4.1	0.4
treeadd	3.7	1.2
tsp	2.0	0.5
voronoi	3.2	0.9

(b) Prefetching Overhead

Figure 16: Performance of greedy prefetching when the number of memory memory units is varied, plus a breakdown of the overhead per prefetch (**1** = 1 memory unit, **2** = 2 memory units, **4** = 4 memory units).

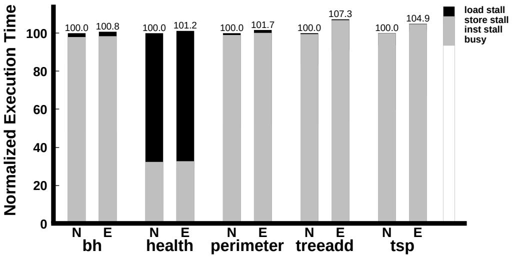
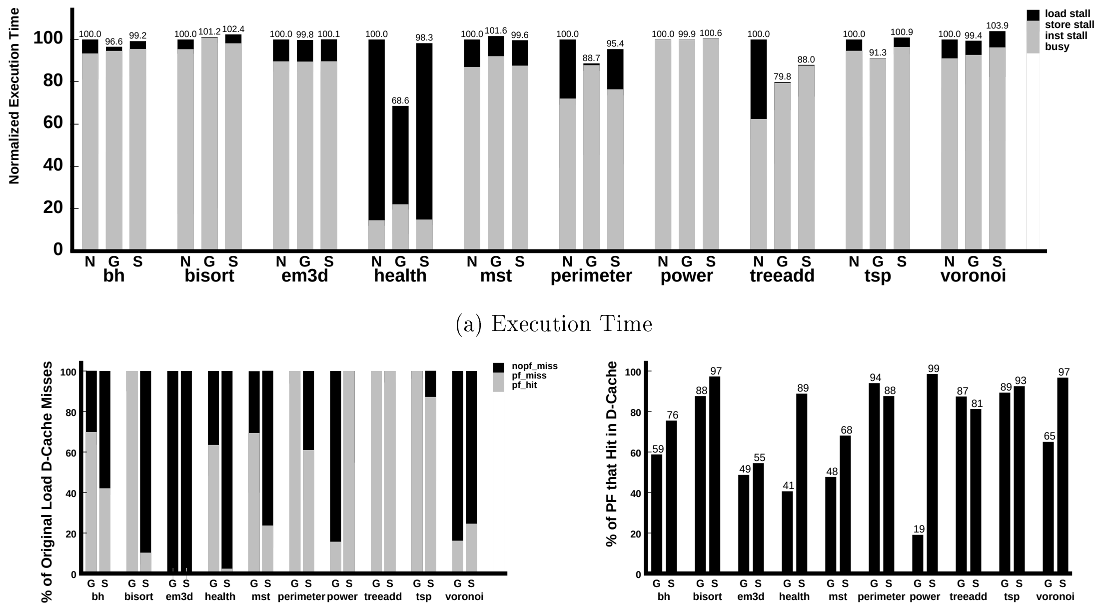


Figure 17: Greedy prefetching with excepting vs. non-excepting prefetches (**N** = non-excepting, **E** = excepting). Execution time is normalized to the **N** case.

MIPS R10000 contains only a single address calculation unit, we ran the experiments presented thus far using two units, since we found this to be important. Figure 16(a) shows the performance when the number of memory units is varied for the five applications that showed significant improvements under greedy prefetching (the "**2**" bars correspond to the "**G**" bars in earlier figures). As we see in Figure 16(a), having two memory units is important, since it improves performance by up to 35% over a single unit. The marginal gain of having four units is considerably smaller.

5.4.2 Support for Non-Excepting Memory Operations

In array-based codes, invalid prefetch addresses typically only occur if one prefetches off the end of an array. In contrast, invalid prefetch addresses may occur frequently in RDS codes due to invalid or NULL pointers. To quantify the benefit of having non-excepting prefetches, we forced the greedy prefetching compiler to enclose any prefetches that may have invalid addresses with a NULL test. As we see in Figure 17, our dynamically-scheduled processor was not able to hide all of this overhead, hence resulting



(a) Execution Time

(b) Coverage Factor

(c) Unnecessary Prefetches

Figure 18: Performance comparison between SPAID and greedy prefetching (**N** = no prefetching, **G** = greedy prefetching, **S** = SPAID).

in up to a 7% increase in execution time. In addition to non-excepting prefetches, non-excepting *load* instructions also appear to be quite useful for prefetching pointer-based codes, although we currently are not exploiting them aggressively in our compiler.

6 Related Work

Although prefetching has been studied extensively for array-based numeric codes [2, 16], relatively little work has been done on non-numeric applications. Chen *et al.* [5] used global instruction scheduling techniques to move address generation back as early as possible to hide a small cache miss latency (10 cycles), and found mixed results. In contrast, our algorithms focus only on RDS accesses, and can issue prefetches much earlier (across procedure and loop iteration boundaries) by overcoming the pointer-chasing problem. Zhang and Torrellas [22] proposed a hardware-assisted scheme for prefetching irregular applications in shared-memory multiprocessors. Under their scheme, programs are annotated to bind together groups of data (e.g., fields in a record or two records linked by a pointer), which are then prefetched under hardware control. Compared to our compiler-based approach, their scheme has two shortcomings: (i) annotations are inserted manually, and (ii) their hardware extensions are not likely to be applicable in uniprocessors.

To our knowledge, the only compiler-based pointer prefetching scheme in the literature is the SPAID scheme proposed by Lipasti *et al.* [14]. Based on an observation that procedures are likely to dereference any pointers passed to them as arguments, SPAID inserts prefetches for the objects pointed to by these pointer arguments at the call sites. Therefore this scheme is only effective if the interval between the start of a procedure call and its dereference of a pointer is comparable to the cache miss latency. To quantify the performance difference between SPAID and our greedy prefetching scheme, we implemented several versions of SPAID in our experimental framework with different numbers of prefetches inserted per call site. Our results are consistent with the conclusion in the SPAID paper [14] that the best performance is achieved by inserting only one prefetch per call site—the **S** bars in Figure 18 correspond to this optimal case. When a procedure has multiple pointer arguments, we select the first one pointing to any RDS to prefetch. We also improved the performance of the proposed SPAID scheme for `treeadd` from a slowdown of 13% to a speedup of 14% by prefetching two cache lines at a time rather than one. As we see in Figure 18, greedy prefetching outperforms SPAID in all cases except `mst`. The problem with SPAID is that it pays significant prefetching overhead without covering many cache misses, as shown by the low coverage factors and high fraction of unnecessary prefetches in Figure 18. In contrast, greedy prefetching does a better job of choosing what to prefetch, and can schedule prefetches earlier to hide more latency.

7 Future Work

Based on the lessons we have learned from these experiments, we are currently extending our research in the following directions. First, we are exploring how to automate history-pointer and data-linearization prefetching in the compiler—and how to automatically choose the best scheme among the three for a given application—to capture the benefits demonstrated in Section 5.2. Second, the results in Section 5.3 suggest that improved prefetching analysis can help to reduce overheads. However, since predicting data locality through static compile-time analysis is difficult—and since feedback-based compilation has its own set of problems—we are exploring the possibility of generating code with prefetching that dynamically adapts to its own memory behavior. Third, our experience with the `mst` application illustrates the difficulty of prefetching hash table accesses, where linked lists are quite short, and the head of the list is data-dependent on the hashing function. To hide the latency in such cases, we must prefetch *before* the hashing function is called—although the overheads of doing so may be significant, it does appear to be feasible. Finally, we are investigating the performance of our schemes in shared-memory multiprocessors, where although the large cache miss latencies increase the potential benefit of prefetching, they also intensify the pointer-chasing problem.

8 Conclusions

While automatic compiler-inserted prefetching has shown considerable success in hiding the memory latency of array-based codes, the compiler technology for successfully prefetching pointer-based data structures has thus far been lacking. In this paper, we propose three prefetching schemes which overcome the pointer-chasing problem, we automate the most widely applicable scheme (greedy prefetching) in the compiler, and we evaluate the performance of all three schemes on a modern superscalar processor similar to the MIPS R10000.

Our experiments show that automatic compiler-inserted prefetching can accelerate pointer-based applications by as much as 45%. In addition, the more sophisticated algorithms (which we currently simulate by hand) can improve performance by as much as twofold. Our experiments also demonstrate the potential benefit of using data locality information to further reduce prefetching overhead.

From an architectural perspective, these encouraging results suggest that the latency problem for pointer-based codes may be addressed largely through the prefetch instructions that already exist in many recent microprocessors. To fully exploit prefetching, our results indicate that an architecture should provide at least two memory units and a non-excepting prefetch instruction. We believe that this work provides a foothold for additional research on compiler-based prefetching for non-numeric applications.

9 Acknowledgments

This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada, and by a grant from IBM Canada's Centre for Advanced Studies. Chi-Keung Luk is partially supported by a Canadian Commonwealth Fellowship. Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[3] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[4] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[5] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of Microcomputing 24*, 1991.

[6] A. Deutsch. A storeless model of aliasing and its abstractions using finite respresentation of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[8] R. Ghiya and L. J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[9] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[10] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 218–229, June 1994.

[11] J. S. Kowalik, editor. *Parallel MIMD Computation : The HEP Supercomputer and Its Applications*. MIT Press, 1985.

[12] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[13] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[14] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, 1995.

[15] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994. Technical Report CSL-TR-94-626.

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[17] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2), March 1995.

[18] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[19] C. J. Stephenson. Fast fits. In *Proceedings of the ACM 9th Symposium on Operating Systems*, October 1983.

[20] S. W. K. Tjiang and J. L. Hennessy. Sharlit: A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[21] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[22] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–200, June 1995.