Automatic Compiler-Inserted Prefetching for Pointer-Based Applications

Chi-Keung Luk and Todd C. Mowry

Abstract— **As the disparity between processor and memory speeds continues to grow, memory latency is becoming an increasingly important performance bottleneck. While software-controlled prefetching is an attractive technique for tolerating this latency, its success has been limited thus far to array-based numeric codes. In this paper, we expand the scope of automatic compiler-inserted prefetching to also include the recursive data structures commonly found in pointer-based applications. We propose three compiler-based prefetching schemes, and automate the most widely applicable scheme (*greedy prefetching*) in an optimizing research compiler. Our experimental results demonstrate that compiler-inserted prefetching can offer significant performance gains on both uniprocessors and large-scale shared-memory multiprocessors.**

Keywords— **Caches, prefetching, pointer-based applications, recursive data structures, compiler optimization, shared-memory multiprocessors, performance evaluation.**

I. Introduction

SOFTWARE -controlled data prefetching [1], [2] offers the potential for bridging the ever-increasing speed gap between the memory subsystem and today's high-performance processors. In recognition of this potential, a number of recent processors have added support for prefetch instructions [3], [4], [5]. While prefetching has enjoyed considerable success in array-based numeric codes [6], its potential in pointer-based applications has remained largely unexplored. This paper investigates compiler-inserted prefetching for pointer-based applications—in particular, those containing 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—there is little inherent spatial locality between consecutively-accessed nodes in an RDS, since they are dynamically allocated at arbitrary addresses.

C.-K. Luk is with the Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3G4, Canada. E-mail: luk@eecg.toronto.edu.

T. C. Mowry is with the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: tcm@cs.cmu.edu.

To cope with the latency of accessing these pointer-based data structures, we propose three compiler-based schemes for prefetching RDSs, as described in Section II. We implemented the most widely-applicable of these schemes—*greedy prefetching*—in a modern research compiler (SUIF [7]), as discussed in Section III. To evaluate our schemes, we performed detailed simulations of their impact on both uniprocessor and multiprocessor systems in Sections IV and V, respectively. Finally, we present related work and conclusions in Sections VI and VII.

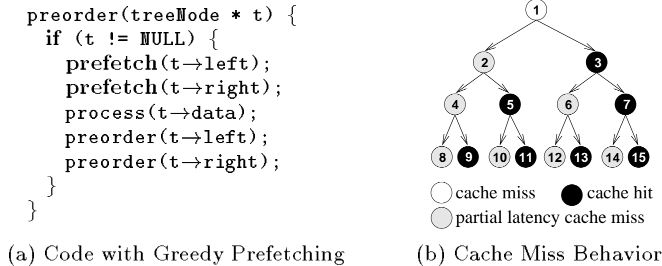
II. Software-Controlled Prefetching for RDSs

A key challenge in successfully prefetching RDSs is scheduling the prefetches sufficiently far in advance to fully hide the latency, while introducing minimal runtime overhead. In contrast with array-based codes, where the prefetching distance can be easily controlled using *software pipelining* [2], the fundamental difficulty with RDSs is that we must first dereference pointers to compute the prefetch addresses. Getting several nodes ahead in an RDS traversal typically involves following a pointer chain. However, the very act of touching these intermediate nodes along the pointer chain means that we cannot tolerate the latency of fetching more than one node ahead.

To overcome this *pointer-chasing problem* [8], we propose three schemes for generating prefetch addresses without following the entire pointer chain. The first two schemes—*greedy prefetching* and *history-pointer prefetching*—use a pointer within the current node as the prefetching address; the difference is that greedy prefetching uses existing pointers, whereas history-pointer prefetching creates new pointers. The third scheme—*data-linearization prefetching*—generates prefetch addresses without pointer dereferences.

A. 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 neighbors can be immediately followed as the next node in the traversal, but there is often a good chance that other neighbors will be visited sometime in the future. Therefore by prefetching all k pointers when a node is first visited, we hope that enough of these



(a) Code with Greedy Prefetching (b) Cache Miss Behavior

Fig. 1. Illustration of greedy prefetching.

prefetches are successful that we can hide at least some fraction of the miss latency.

To illustrate how greedy prefetching works, consider the pre-order traversal of a binary tree (i.e. $k = 2$), where Figure 1(a) shows the code with greedy prefetching added. Assuming that the computation in `process()` takes half as long as the cache miss latency L, we would want to prefetch two nodes ahead to fully hide the latency. Figure 1(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 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).

Greedy prefetching offers the following advantages: (i) it has low runtime overhead, since no additional storage or computation is needed to construct the prefetch 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 III. 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.

B. History-Pointer Prefetching

Rather than relying on existing pointers to approximate prefetch addresses, we can potentially synthesize more accurate pointers based on the observed RDS traversal patterns. To prefetch d nodes ahead under the *history-pointer prefetching* scheme [8], we add a new pointer (called a *history-pointer*) to a node n_i to record the observed address of n_{i+d} (the node visited d nodes after n_i) on a recent traversal of the RDS. 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. 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.

In contrast with greedy prefetching, history-pointer prefetching offers no improvement on the first traversal of an RDS, but can potentially hide all of the latency on subsequent traversals. While history-pointer prefetching offers the potential advantage of improved latency tolerance, this comes at the expense of (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 RDS structure changes. In one extreme, if the RDS never changes, we can set the history-pointers just 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.

C. Data-Linearization Prefetching

The idea behind *data-linearization prefetching* [8] 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 generate prefetch addresses and launch them early enough. Another advantage of this scheme is that it improves spatial locality. The major challenge, however, is how and when we can generate this data layout. 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. Instead, the easiest 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). If the RDS does change radically, the program will still behave correctly, but prefetching will not improve performance.

III. Implementation of Greedy Prefetching

Of the three schemes that we propose, 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 within the SUIF compiler [7], and we will simulate the other two algorithms by hand. Our implementation consists of an *analysis* phase to recognize RDS accesses, and a *scheduling* phase to insert prefetches.

A. 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

```
struct T {            struct A {              struct C {
  int data;             int i;                  int j;
  struct T *left;       struct B **kids[8];     double f;
  struct T *right;    }                       }
}
```

(a) RDS type (b) RDS type (c) Not RDS type

Fig. 2. Examples of which types are recognized as RDS types.

```
while (1)          for (...)       f(tree *t)        k(tree tn)
{                  {               {                 {
  list *m;           list *n;        ...               ...
  ...                ...             f(t→left);        k(*(tn.left));
  m = l→next;        n = g(n);       f(t→right);       k(*(tn.right));
  l = m→next;        ...             ...               ...
  ...              }               }                 }
}
```

(a) (b) (c) (d)

Fig. 3. Examples of control structures recognized as RDS traversals.

be comprised of heterogeneous nodes.) For example, the type declarations in Figure 2(a) and Figure 2(b) would be recognized as RDS types, whereas Figure 2(c) would not.

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 [9], [10].

Figure 3 shows some example program fragments that our compiler treats as RDS accesses. In Figure 3(a), `l` is updated to `l→next→next` inside the while-loop. In Figure 3(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 3(c), two dereferences of the function argument `t` are passed as the parameters to two recursive calls. Figure 3(d) is similar to Figure 3(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 to eliminate unnecessary prefetches. Although we have not automated this step in our compiler, we evaluated its potential benefits in an earlier study [8].

B. Scheduling Prefetches

Once RDS accesses have been recognized, the compiler inserts greedy prefetches as follows. At the point where an RDS object 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 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 4. Further details of our implementation can be found in Luk's thesis [11].

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

(a) Loop

```
f(tree *t) {                     f(tree *t) {
  tree *q;             ⟹          tree *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

Fig. 4. Examples of greedy prefetch scheduling.

TABLE I
Benchmark characteristics.

Benchmark	Recursive Data Structures Used	Input Data Set	Node Memory Allocated
BH	Heterogeneous octree	4K bodies	721 KB
Bisort	Binary tree	250,000 integers	1,535 KB
EM3D	Singly-linked lists	2000 H-nodes, 100 E-nodes, 75% local	1,671 KB
Health	Four-way tree and doubly-linked lists	level = 5, time = 500	925 KB
MST	Array of singly-linked lists	512 nodes	10 KB
Perimeter	A quadtree	4Kx4K image	6,445 KB
Power	Multi-way tree and singly-linked lists	10,000 customers	418 KB
TreeAdd	Binary tree	1024K nodes	12,288 KB
TSP	Binary tree and doubly-linked lists	100,000 cities	5,120 KB
Voronoi	Binary tree	20,000 points	10,915 KB

IV. Prefetching RDSs on Uniprocessors

In this section, we quantify the impact of our prefetching schemes on *uniprocessor* performance. Later, in Section V, we will turn our attention to multiprocessor systems.

A. Experimental Framework

We performed detailed cycle-by-cycle simulations of the entire Olden benchmark suite [12] on a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [5]. The Olden benchmark suite contains ten pointer-based applications written in C, which are briefly summarized in Table I. The rightmost column in Table I shows the amount of memory dynamically allocated to RDS nodes.

Our simulation model varies slightly from the actual MIPS R10000 (e.g., we model two memory units, and we

TABLE II
UNIPROCESSOR 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

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. Table II shows the parameters of our model. We use *pixie* [13] to instrument the optimized MIPS object files produced by the compiler, and pipe the resulting trace into our simulator.

To avoid misses during the initialization of dynamically-allocated objects, we used a modified version of the IRIX `mallopt` routine [14] whereby we prefetch allocated objects before they are initialized. Determining these prefetch addresses is straightforward, since objects of the same size are typically allocated from contiguous memory. This optimization alone led to over twofold speedups relative to using `malloc` for the majority of the applications—particularly those that frequently allocate small objects.

B. Performance of Greedy Prefetching

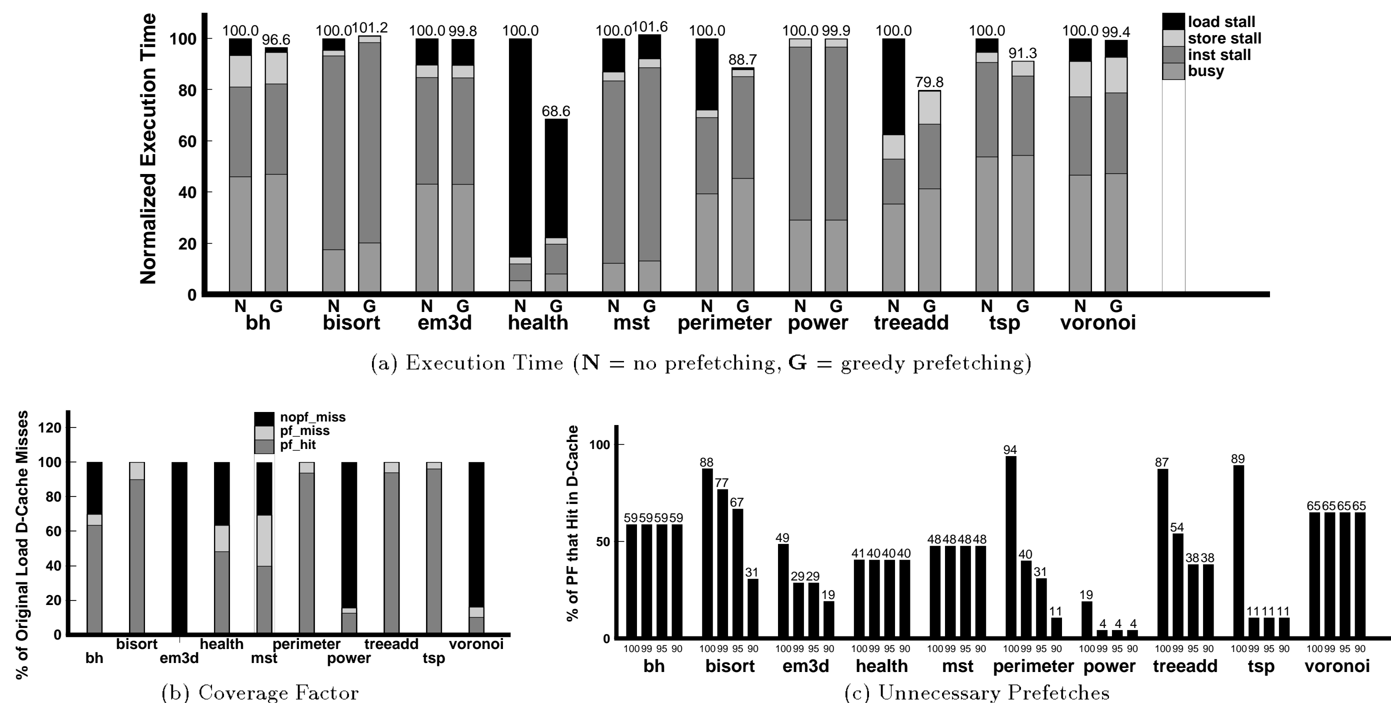
Figure 5 shows the results of our uniprocessor experiments. The overall performance improvement offered by greedy prefetching is shown in Figure 5(a), where the two bars correspond to the cases without prefetching (**N**) and with greedy prefetching (**G**). These bars represent execution time normalized to the case without prefetching, and they are broken down into four categories explaining what happened during all potential graduation slots. (The number of graduation slots is the issue width—4 in this case—multiplied by the number of cycles.) 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, 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 misses, since these delays also exacerbate subsequent data dependence stalls.

As we see in Figure 5(a), half of the applications enjoy a speedup ranging from 4% to 45%, and 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.

To understand the performance results in greater depth, Figure 5(b) 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 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.

To help evaluate the costs of prefetching, Figure 5(c) 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 found that eliminating prefetches with hit rates above 95% improves performance by 1-7% for these applications [8].

Finally, we measured the impact of greedy prefetching on memory bandwidth consumption. We observe that on av-



(a) Execution Time (**N** = no prefetching, **G** = greedy prefetching)

(b) Coverage Factor

(c) Unnecessary Prefetches

Fig. 5. Performance impact of compiler-inserted greedy prefetching on a uniprocessor.

erage, greedy prefetching increases the traffic between the primary and secondary caches by 12.7%, and the traffic between the secondary cache and main memory by 7.8%. In our experiments, this has almost no impact on performance. Hence greedy prefetching does not appear to be suffering from memory bandwidth problems.

In summary, we have seen that automatic compiler-inserted prefetching can result in significant speedups for uniprocessor applications containing RDSs. We now investigate whether the two more sophisticated prefetching schemes can offer even larger performance gains.

C. Performance of History-Pointer Prefetching and Data-Linearization Prefetching

We applied history-pointer prefetching and data-linearization prefetching by hand to several of our applications. History-pointer prefetching is applicable to `health` because the list structures that are accessed by a key procedure remain unchanged across the over ten thousand times that it is called. As a result, history-pointer prefetching achieves a 40% speedup over greedy prefetching through better miss coverage and fewer unnecessary prefetches. Although history-pointer prefetching has fewer unnecessary prefetches than greedy prefetching, it has significantly higher instruction overhead due to the extra work required to maintain the history-pointers.

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). By reducing the number of unnecessary prefetches (and hence prefetching overhead) while maintaining good coverage factors, data-linearization prefetching results in speedups of 9% and 18% over greedy prefetching for `perimeter` and `treeadd`, respectively. Overall, we see that both schemes can potentially offer significant improvements over greedy prefetching when applicable.

V. Prefetching RDSs on Multiprocessors

Having observed the benefits of automatic prefetching of RDSs on *uniprocessors*, we now investigate whether the compiler can also accelerate pointer-based applications running on *multiprocessors*. In earlier studies, Mowry demonstrated that the compiler can successfully prefetch parallel *matrix-based* codes [2], [15], but the compiler used in those studies did not attempt to prefetch *pointer-based* access patterns. However, through hand-inserted prefetching, Mowry was able to achieve a significant speedup in `BARNES` [15], which is a pointer-intensive shared-memory parallel application from the SPLASH suite [16].

`BARNES` performs a hierarchical n-body simulation of the evolution of galaxies. The main computation consists of a depth-first traversal of an octree structure to compute the gravitational force exerted by the given body on all other bodies in the tree. This is repeated for each body in the system, and the bodies are statically assigned to processors for the duration of each time step. Cache misses occur whenever a processor visits a part of the octree that is not already in its cache, either due to replacements or communication. To insert prefetches by hand, Mowry used a strategy similar to greedy prefetching: upon first arriving at a node, he prefetched all immediate children before descending depth-first into the first child.

TABLE III
MEMORY LATENCIES IN MULTIPROCESSOR SIMULATIONS.

Destination of Access	Read	Write
Primary Cache	1 cycle	1 cycle
Secondary Cache	15 cycles	4 cycles
Local Node	29 cycles	17 cycles
Remote Node	101 cycles	89 cycles
Dirty Remote, Remote Home	132 cycles	120 cycles

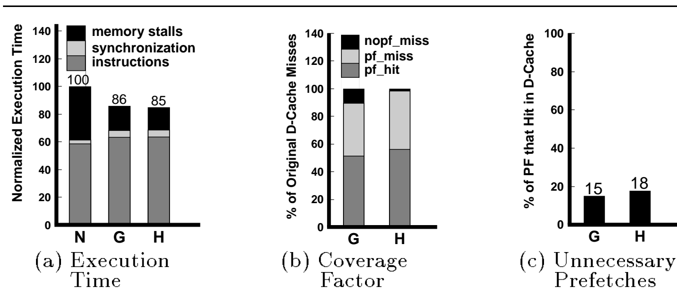


Fig. 6. Impact of compiler-inserted greeding prefetching on BARNES on a multiprocessor (**N** = no prefetching, **G** = compiler-inserted greedy prefetching, **H** = hand-inserted prefetching).

To evaluate the performance of our compiler-based implementation of greedy prefetching on a multiprocessor, we compared it with hand-inserted prefetching for BARNES. For the sake of comparison, we adopted the same simulation environment used in Mowry's earlier study [15], which we now briefly summarize. We simulated a cache-coherent, shared-memory multiprocessor that resembles the DASH multiprocessor [17]. Our simulated machine consists of 16 processors, each of which has two levels of direct-mapped caches, both using 16 byte lines. Table III shows the latency for servicing an access to different levels of the memory hierarchy, in the absence of contention (our simulations did model contention, however). To make simulations feasible, we scaled down both the problem size and cache sizes accordingly (we ran 8192 bodies through 3 times steps on an 8K/64K cache hierarchy), as was done (and explained in more detail) in the original study [2].

Figure 6 shows the impact of both compiler-inserted greedy prefetching (**G**) and hand-inserted prefetching (**H**) on BARNES. The execution times in Figure 6(a) are broken down as follows: the bottom section is the amount of time spent executing instructions (including any prefetching instruction overhead), and the middle and top sections are synchronization and memory stall times, respectively. As we see in Figure 6(a), the compiler achieves nearly identical performance to hand-inserted prefetching. The compiler prefetches 90% of the original cache misses with only 15% of these misses being unnecessary, as we see in Figures 6(b) and 6(c), respectively. Of the prefetched misses, the latency was fully hidden in half of the cases (*pf_hit*), and partially hidden in the other cases (*pf_miss*). By eliminating roughly half of the original memory stall time, the compiler was able to achieve a 16% speedup.

The compiler's greedy strategy for inserting prefetches is quite similar to what was done by hand, with the following exception. In an effort to minimize unnecessary prefetches, the compiler's default strategy is to prefetch only the first 64 bytes within a given RDS node. In the case of BARNES, the nodes are longer than 64 bytes, and we discovered that hand-inserted prefetching achieves better performance when we prefetch the entire nodes. In this case, the improved miss coverage of prefetching the entire nodes is worth the additional unnecessary prefetches, thereby resulting in a 1% speedup over compiler-inserted prefetching. Overall, however, we are quite pleased that the compiler was able to do this well, nearly matching the best performance that we could achieve by hand.

VI. Related Work

Although prefetching has been studied extensively for array-based numeric codes [6], [18], relatively little work has been done on non-numeric applications. Chen *et al.* [19] 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 [20] 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 with 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. Joseph and Grunwald [21] proposed a hardware-based Markov prefetching scheme which prefetches multiple predicted addresses upon a primary cache miss. While Markov prefetching can potentially handle chaotic miss patterns, it requires considerably more hardware support and has less flexibility in selecting what to prefetch and controlling the prefetch distance than our compiler-based schemes.

To our knowledge, the only compiler-based pointer prefetching scheme in the literature is the SPAID scheme proposed by Lipasti *et al.* [22]. 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. In an earlier study [8], we found that greedy prefetching offers substantially better performance than SPAID by hiding more latency while paying less overhead.

VII. 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 its performance on both a modern superscalar uniprocessor (similar to the MIPS R10000) and on a large-scale shared-memory multiprocessor. Our uniprocessor 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 offer even larger performance gains. Our multiprocessor experiments demonstrate that the compiler can potentially provide equivalent performance to hand-inserted prefetching even on parallel applications. 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.

Acknowledgments

This work is supported 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] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 40–52.

[2] T. C. Mowry, *Tolerating Latency Through Software-Controlled Data Prefetching*, Ph.D. thesis, Stanford University, March 1994.

[3] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan, "Compiler techniques for data prefetching on the PowerPC," in *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, June 1995, pp. 19–26.

[4] V. Santhanam, E. Gornish, and W.-C. Hsu, "Data prefetching on the HP PA8000," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 264–273.

[5] K. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, pp. 28–41, April 1996.

[6] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 62–73.

[7] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, Dec 1994.

[8] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. 222–233.

[9] 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*, June 1994, pp. 242–256.

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

[11] C.-K. Luk, *Optimizing the Cache Performance of Non-Numeric Applications*, Ph.D. thesis, Department of Computer Science, University of Toronto, forthcoming.

[12] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM Transactions. on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, March 1995.

[13] M. D. Smith, "Tracing with pixie," Tech. Rep. CSL-TR-91-497, Stanford University, November 1991.

[14] C. J. Stephenson, "Fast fits," in *Proceedings of the ACM 9th Symposium on Operating Systems*, October 1983, pp. 30–32.

[15] T. C. Mowry, "Tolerating latency in multiprocessors through compiler-inserted prefetching," *ACM Transactions on Computer Systems*, vol. 16, no. 1, pp. 55–92, 1998.

[16] J. P. Singh, W.-D. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared memory," Tech. Rep. CSL-TR-91-469, Stanford University, April 1991.

[17] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, March 1992.

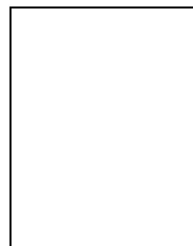
[18] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of Supercomputing '91*, 1991, pp. 176–186.

[19] 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 the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, 1991, pp. 69–73.

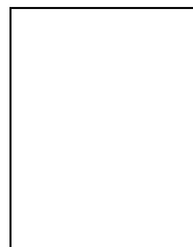
[20] Z. Zhang and J. Torrellas, "Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 188–200.

[21] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 252–263.

[22] 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 ACM/IEEE International Symposium on Microarchitecture*, 1995, pp. 231–236.



Chi-Keung Luk is a Ph.D. candidate in the Department of Computer Science at the University of Toronto, and is currently a visiting scholar at Carnegie Mellon University. He received his B.Sc. (First Class Honors) and M.Phil. degrees in computer science, both from The Chinese University of Hong Kong. His research interests are computer architecture, compiler optimizations, and programming languages, with a focus on the memory performance of non-numeric applications. He has been awarded a Canadian Commonwealth Fellowship, an IBM CAS Fellowship, and a Croucher Foundation Fellowship. Further information about his current research activities can be found at http://www.cs.cmu.edu/~luk.



Todd C. Mowry received his B.S.E.E. from the University of Virginia in 1988, and his M.S.E.E. and Ph.D. from Stanford University in 1989 and 1994, respectively. From 1994 through 1997, he was an assistant professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto. Since 1997, he has been an associate professor in the Computer Science Department at Carnegie Mellon University. Dr. Mowry's research interests span architecture, compilers, and operating systems. Most recently, he has been focusing on automatically tolerating the latency of accessing and communicating data, and on automatically extracting thread-level parallelism from non-numeric applications. Further information about his current research activities can be found at http://www.cs.cmu.edu/~tcm.