Push vs. Pull: Data Movement for Linked Data Structures*

Chia-Lin Yang and Alvin R. Lebeck

Department of Computer Science
Duke University
Durham, NC 27708 USA
{yangc,alvy}@cs.duke.edu

Abstract

As the performance gap between the CPU and main memory continues to grow, techniques to hide memory latency are essential to deliver a high performance computer system. Prefetching can often overlap memory latency with computation for array-based numeric applications. However, prefetching for pointer-intensive applications still remains a challenging problem. Prefetching linked data structures (LDS) is difficult because the address sequence of LDS traversal does not present the same arithmetic regularity as array-based applications and the data dependence of pointer dereferences can serialize the address generation process.

In this paper, we propose a cooperative hardware/software mechanism to reduce memory access latencies for linked data structures. Instead of relying on the past address history to predict future accesses, we identify the load instructions that traverse the LDS, and execute them ahead of the actual computation. To overcome the serial nature of the LDS address generation, we attach a prefetch controller to each level of the memory hierarchy and push, rather than pull, data to the CPU. Our simulations, using four pointer-intensive applications, show that the push model can achieve between 4% and 30% larger reductions in execution time compared to the pull model.

1. INTRODUCTION

Since 1980, microprocessor performance has improved 60% per year, however, memory access time has improved only 10% per year. As the performance gap between processors and memory continues to grow, techniques to reduce the effect of this disparity are essential to build a high performance computer system. The use of caches between the CPU and main memory is recognized as an effective method to bridge this gap [21]. If programs exhibit good temporal and spatial locality, the majority of memory requests can be satisfied by caches without having to access main memory. However, a cache's effectiveness can be limited for programs with poor locality.

Prefetching is a common technique that attempts to hide memory latency by issuing memory requests earlier than demand fetching a cache block when a miss occurs. Prefetching works very well for programs with regular access patterns. Unfortunately, many applications create sophisticated data structures using pointers, and do not exhibit sufficient regularity for conventional prefetch techniques to exploit. Furthermore, prefetching pointer-intensive data structures can be limited by the serial nature of pointer dereferences—known as the pointer chasing problem—since the address of the next node is not known until the contents of the current node is accessible.

For these applications, performance may improve if lower levels of the memory hierarchy could dereference pointers and pro-actively *push* data up to the processor rather than requiring the processor to *fetch* each node before initiating the next request. An oracle could accomplish this by knowing the layout and traversal order of the pointer-based data structure, and thus overlap the transfer of consecutively accessed nodes up the memory hierarchy. The challenge is to develop a realizable memory hierarchy that can obtain these benefits.

This paper presents a novel memory architecture to perform pointer dereferences in the lower levels of the memory hierarchy and push data up to the CPU. In this architecture, if the cache block containing the data corresponding to pointer p (*p) is found at a particular level of the memory hierarchy, the request for the next element (*p+offset) is issued from that level. This allows the access for the next node to overlap with the transfer of the previous element. In this way, a series of pointer dereferences becomes a pipelined process rather than a serial process.

In this paper, we present results using specialized hardware to support the above push model for data movement in list-based programs. Our simulations using four pointer-intensive benchmarks show that conventional prefetching using the pull model can achieve speedups of 0% to 223%, while the new push model achieves speedups from 6% to 227%. More specifically, the push model reduces execution time by 4% to 30% more than the pull model.

*This work supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637, CDA-95-12356, and EIA-99-72879, Career Award MIP-97-02547, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

The rest of this paper is organized as follows. Section 2 provides background information and motivation for the push model. Section 3 describes the basic push model and details of a proposed architecture. Section 4 presents our simulation results. Related work is discussed in Section 5 and we conclude in Section 6.

2. BACKGROUND AND MOTIVATION

Pointer-based data structures, often called linked data structures (LDS), present two challenges to any prefetching technique: address irregularity and data dependencies. To improve the memory system performance of many pointer-based applications, both of these issues must be addressed.

2.1 Address Irregularity

Prefetching LDS is unlike array-based numeric applications since the address sequence from LDS accesses does not have arithmetic regularity. Consider traversing a linear array of integers versus traversing a linked-list. For linear arrays, the address of array elements accessed at iteration i is $A_{addr} + 4 * i$, where A_{addr} is the base address of array A and i is the iteration number. Using this formula, the address of element A[i] at any iteration i is easily computed. Furthermore, the above formula produces a compact representation of array accesses, thus making it easier to predict future addresses. In contrast, LDS elements are allocated dynamically from the heap and adjacent elements are not necessarily contiguous in memory. Therefore, traditional array-based address prediction techniques are not applicable to LDS accesses.

Instead of predicting the future LDS addresses based on address regularity, Roth et. al [19] observed that the load instructions (Program Counters) that access LDS elements are themselves a compact representation of LDS accesses—called a *traversal kernel.* Consider the linked-list traversal example shown in Figure 1, the structure *list* is the LDS being traversed. Instructions 1 and 3 access *list* itself, while instruction 2 accesses another portion of the LDS through pointer p. These three instructions make up the list traversal kernel. Assembly code for this kernel is shown in Figure 1b.

Consecutive iterations of the traversal kernel create data dependencies. Instruction 3 at iteration a produces an address consumed by itself and instruction 1 in the next iteration b. Instruction 1 produces an address consumed by instruction 2 in the same iteration. This relation can be represented as a dependency tree, shown in Figure 1c. Loads in LDS traversal kernels can be classified into three types: *recurrent*, *traversal*, and *data*. Instruction 3 (list=list→forward) generates the address of the next element of the *list* structure, and is classified as a *recurrent* load. Instruction 1 (p=list→patient) produces addresses consumed by other pointer loads and is called a *traversal* load. All the loads in the leaves of the dependency tree, such as instruction 2 (p→time++) in this example, are *data* loads.

Roth et. al proposed dynamically constructing this kernel and repeatedly executing it independent of the CPU execution. This method overcomes the irregular address sequence problem. However, its effectiveness is limited because of the serial nature of LDS address generation.

2.2 Data Dependence

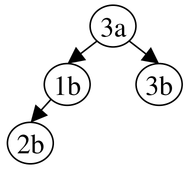
The second challenge for improving performance for LDS traversal is to overcome the serialization caused by data dependencies. In LDS traversal a series of pointer dereferences is required to generate future element addresses. If the address of node i is stored in the pointer p, then the address of node i+1 is *(p+x), where x is the offset of the next field. The address *(p+x) is not known until the value of p is known. This is commonly called the pointer-chasing problem. In this scenario, the serial nature of the memory access can make it difficult to hide memory latencies longer than one iteration of computation.

```
list = head;
while(list != NULL) {
  p = list->patient; /* (1) traversal load */
  x = p->data; /* (2) data load */
  list = list->forward; /* (3) recurrent load */
}
```

a) source code

```
1: lw $15, o1($24)
2: lw $4,  o2($15)
3: lw $24, o3($24)
```

b) LDS Traversal Kernel



c) Data Dependencies

Figure 1: Linked Data Structure (LDS) Traversal

The primary contribution of this paper is the development of a new model for data movement in linked data structures that reduces the impact of data dependencies. By abandoning the conventional *pull* approach of initiating all memory requests (demand fetch or prefetch) by the processor or upper level of the memory hierarchy, we can eliminate some of the delay in traversing linked data structures. The following section elaborates on the new model and a proposed implementation.

3. THE PUSH MODEL

Most existing prefetch techniques issue requests from the CPU level to fetch data from the lower levels of the memory hierarchy. Since pointer-dereferences are required to generate addresses for successive prefetch requests (recurrent loads), these techniques serialize the prefetch process. The prefetch schedule is shown in Figure 2(a), assuming each recurrent load is a L2 cache miss. The memory latency is divided into six parts (r1: sending a request from L1 to L2; a1: L2 access; r2: sending a request from L2 to main memory; a2: memory access; x2: transferring a cache block back to L2; x1: transferring a cache block back to L1). Using this timing, node i+1 arrives at the CPU (r1+a1+r2+a2+x2+x1) cycles (the full memory access latency) after node i.

In the push model, pointer dereferences are performed at the lower levels of the memory hierarchy by executing

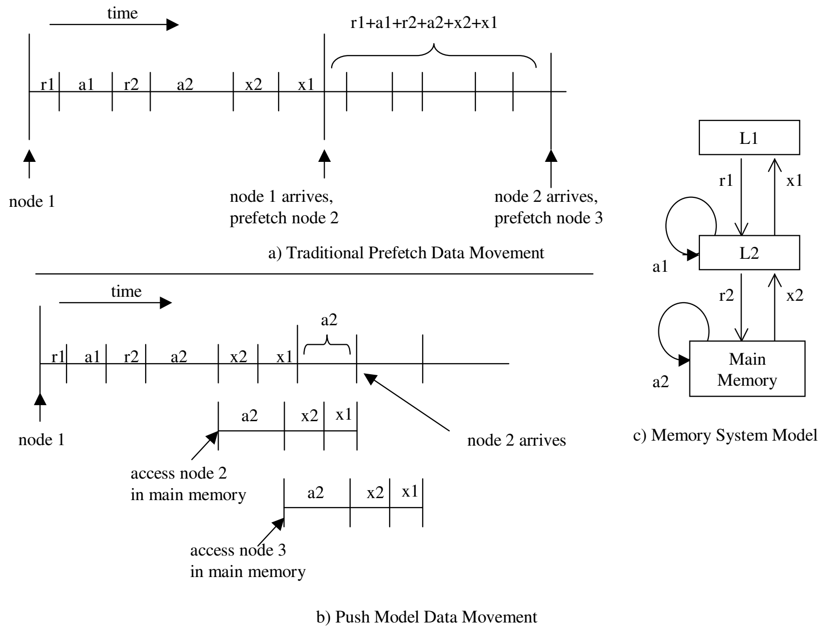


Figure 2: Data Movement for Linked Data Structures

traversal kernels in micro-controllers associated with each memory hierarchy level. This eliminates the request from the upper levels to the lower level, and allows us to overlap the data transfer of node i with the RAM access of node i+1. For example, assume that the request for node i accesses main memory at time t. The result is known to the memory controller at time t+a2, where a2 is the memory access latency. The controller then issues the request for node 2 to main memory at time t+a2, thus overlapping this access with the transfer of node 1 back to the CPU.

This process is shown in Figure 2(b) (Note: in this simple model, we ignore the possible overhead for address translation. Section 4 examines this issue). In this way, we are able to request subsequent nodes in the LDS much earlier than a traditional system. In the push architecture, node i+1 arrives at the CPU a2 cycles after node i. From the CPU standpoint, the latency between node accesses is reduced from (r1+a1+r2+a2+x2+x1) to a2. Assuming 100-cycle overall memory latency and 60-cycle memory access time, the total latency is potentially reduced by 40%. The actual speedup for real applications depends on (i) the contribution of LDS accesses to total memory latency, (ii) the percentage of LDS misses that are L2 misses and (iii) the amount of computation in the program that can be overlapped with the memory latency.

In the push model, a controller—called a prefetch engine—is attached to each level of the memory hierarchy, as shown in Figure 3. The prefetch engines (PFE) executes traversal kernels independent of CPU execution. Cache blocks accessed by the prefetch engines in the L2 or main memory levels are pushed up to the CPU and stored either in the L1 cache or a prefetch buffer. The prefetch buffer is a small, fully-associative cache, which can be accessed in parallel with the L1 cache. The remainder of this section describes the design details of this novel push architecture.

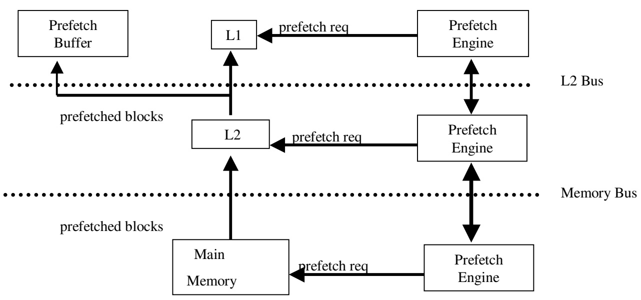


Figure 3: The Push Architecture

3.1 A Push Model Implementation

The function of the prefetch engine is to execute traversal kernels. In this paper we focus on a prefetch engine designed to traverse linked-lists. Support for more sophisticated data structures (e.g., trees) is an important area that we are currently investigating, but it remains future work.

Consider the linked-list traversal example in Figure 1. The prefetch engine is designed to execute iterations of kernel loops until the returned value of the recurrent load (instruction 3: list=list→forward) is NULL. Instances of the recurrent load (3a,3b,3c,.. etc) form a dependence chain which is the critical path in the kernel loop. To allow the prefetch engine to move ahead along this critical path as fast as possible, the prefetch engine must support out-of-order execution. General CPUs implement complicated register renaming and wakeup logic [14; 22] to achieve out-of-order execution. However, the complexity and cost of these techniques is too high for our prefetch engine design.

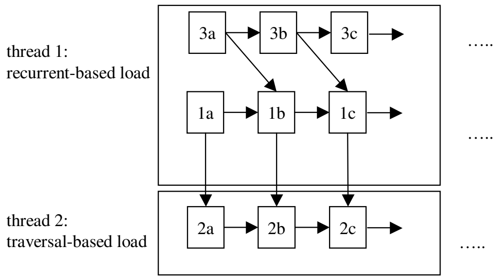


Figure 4: Kernel Execution

To avoid complicated logic, the prefetch engine executes a single traversal kernel in two separate threads, as shown in Figure 4. The first thread executes *recurrent-based* loads, which are dependent on recurrent loads, (e.g., inst. 1: p = list→patient and inst. 3: list = list→forward). This thread executes in order and starts a new iteration once a recurrent load completes. The other thread executes *traversal-based* loads, which are loads that depend on traversal loads, such as inst. 2 (p→time++). Because traversal loads from different iterations, (e.g., 1a and 1b) are allowed to complete out of order, since they can hit in different levels of the memory hierarchy, traversal-based loads should be able to execute out of order.

The internal structure of the prefetch engine is shown in Figure 5. There are four main hardware structures in the prefetch engine: 1) Recurrent Buffer (RB), 2) Producer-Consumer Table (PCT), 3) Base Address Buffer (BAB), and 4) Prefetch Request Queue. The remainder of this section describes its basic operation.

The Recurrent Buffer (RB) contains the type, PC, and offset of recurrent-based loads. The type field indicates if the corresponding load is a recurrent load. Traversal-based loads are stored in the Producer-Consumer Table (PCT). This structure is similar to the Correlation Table in [19], it contains the load's offset, PC and the producer's PC. The PCT is indexed using the producer PC.

We initialize the prefetch engine by down-loading a traversal kernel to the RB and PCT through a memory mapped interface. Prior to executing an LDS access (e.g., list traversal), the program starts the PFE by storing the root address of the LDS in the Base Address Buffer (BAB). We accomplish this by assuming a "flavored" load that annotates a load with additional information, in this case that the data being loaded is the root address of an LDS. When the PFE detects that the BAB is not empty and the instruction queue is empty, it loads the first element of the BAB into the Base register and fetches the recurrent-based instructions from the RB into the instruction queue. At each cycle, the first instruction of the instruction queue is processed, and a new prefetch request is formed by adding the instruction's offset to the value in the Base register. The prefetch request is then stored in the prefetch request queue (PRQ) along with its type and PC.

When a traversal load completes, its PC is used to search the PCT. For each matching entry, the effective address of the traversal-based load is generated by adding its offset to the result of the probing traversal load. In this way, the prefetch engine allows the recurrent-based and traversal-based threads to execute independently.

We assume that each level of the memory hierarchy has a small buffer that stores requests from the CPU, PFE, and cache coherence transactions (e.g., snoop requests) if appropriate. By maintaining information on the type of access, the cache controller can provide the appropriate data required for the PFE to continue execution. For example, if the access is a recurrent prefetch, the resulting data (the address of the next node) is extracted and stored into the BAB. If the access is a traversal load prefetch, the PC is used to probe the PCT and the resulting data is added to the offset from the PCT entry. Before serving any request, the controller checks if a request to the same cache block already exists in the request buffer. If such a request exists, the controller simply drops the new request.

Our data movement model also places new demands on the memory hierarchy to correctly handle cache blocks that were not requested by the processor. In particular, when all memory requests initiate from the CPU level, a returning cache block is guaranteed to find a corresponding MSHR entry that contains information about the request, and provides buffer space for the data if it must compete for cache ports. In our model, it is no longer guaranteed that an arriving cache block will find a corresponding MSHR. One could exist if the processor executed a load for the corresponding cache block. However, in our implementation, if there is no matching MSHR the returned cache block can be stored in a free MSHR or in the request buffer. If neither one is available, this cache block is dropped. Dropping these cache blocks does not affect the program correctness, since they are a result of a non-binding prefetch operation, not a demand fetch from the processor. Note that the push model requires the transferred cache blocks to also transfer their addresses up the memory hierarchy. This is similar to a coherence snoop request traversing up the memory hierarchy.

3.2 Interaction Among Prefetch Engines

Having discussed the design details of individual prefetch engines, we now address the interaction between prefetch engines at the various levels. Recall, that the root address of a LDS is conveyed to the prefetch engine through a special instruction, which we call *ldroot*. If the memory access of a *ldroot* instruction is a hit in the L1 cache, the loaded value is stored in the BAB of the prefetch engine in the fist level. This triggers the prefetching activity in the L1 prefetch engine.

If a *ldroot* instruction or the prefetch for a recurrent load causes a L1 cache miss, the cache miss is passed down to the L2 like other cache misses but with the type bits set to RP (recurrent prefetch). If this miss is satisfied by the L2, the cache controller will detect the RP type and store the result in the BAB. The prefetch engine in the L2 level will find out that the BAB is not empty and start prefetching. If the L1 cache miss is a result of a recurrent load prefetch, the L1 PFE stops execution and the L2 PFE begins execution. Data provided by the L2 PFE is placed in the prefetch buffer to avoid polluting the L1 data cache.

The interaction between the L2 PFE and the memory level PFE is similar the the L1/L2 PFE interaction. The L2 PFE suspends execution when it incurs a miss on a recurrent load, and the memory PFE begins execution when it

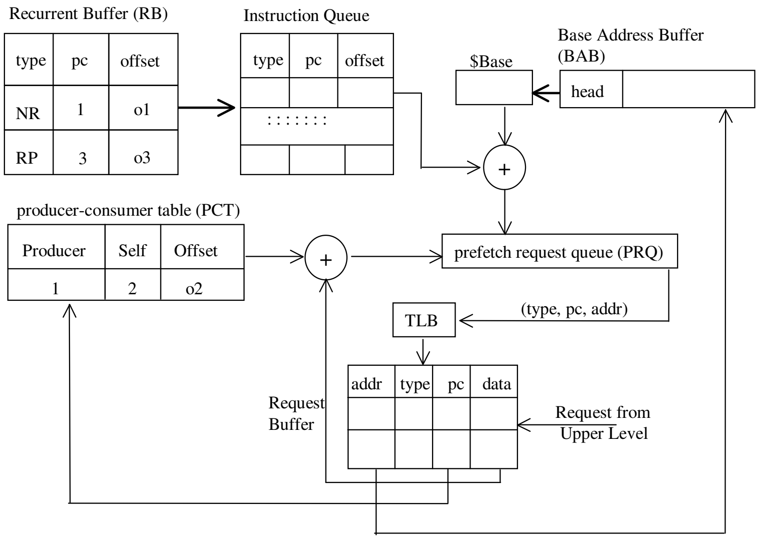


Figure 5: The Prefetch Engine

observes a non-empty BAB. Since the data provided by the memory PFE passes the L2 cache, we deposit the data in both the L2 cache and the prefetch buffer. The L2 cache is generally large enough to contain the additional data blocks and not suffer from severe pollution effects.

4. EVALUATION

This section presents simulation results that demonstrate the effectiveness of the push model for a set of four list-based benchmarks. First, we describe our simulation framework. This is followed by an analysis of a microbenchmark for both the push model and the conventional pull model for prefetching. We finish by evaluating the two models for four macrobenchmarks and examining the impact of address translation overhead.

4.1 Methodology

To evaluate the push model we modified the SimpleScalar simulator [3] to also simulate our prefetch engine implementation. SimpleScalar models a dynamically scheduled processor using a Register Update Unit (RUU) and a Load/Store Queue (LSQ) [22]. The processor pipeline stages are: 1) Fetch: Fetch instructions from the program instruction stream, 2) Dispatch: Decode instructions, allocate RUU, LSQ entries, 3) Issue/Execute: Execute ready instructions if the required functional units are available, 4) Writeback: Supply results to dependent instructions, and 5) Commit: Commit results to the register file in program order, free RUU and LSQ entries.

Our base processor is a 4-way superscalar, out-of-order processor. The memory system consists of a 32KB, 32-byte cache line, 2-way set-associative first level data cache and a 512KB, 64-byte cache line, 4-way set associative second level cache. Both are lock-up free caches that can have up to eight outstanding misses. The L1 cache has 2 read/write ports, and can be accessed in a single cycle. We also simulate a 32-entry fully-associative prefetch buffer with four read/write ports, that can be accessed in parallel with the L1 data cache. The second level cache is pipelined with an 18 cycle round-trip latency. Latency to main memory after an L2 miss is 66 cycles. The CPU has 64 RUU entries and a 16 entry load-store queue, a 2-level branch predictor with a total of 8192 entries, and a perfect instruction cache.

We simulate the pull model by using only a single prefetch engine attached to the L1 cache. This engine executes the same traversal kernels used by the push architecture. However, prefetch requests that miss in the L1 cache must wait for the requested cache block to return from the lower levels before the fetch for the next node can be initiated.

4.2 Microbenchmark Results

We begin by examining the performance of both prefetching models (push and pull) using a microbenchmark. Our microbenchmark simply traverses a linked list of 32,768 32-byte nodes 10 times, see Figure 6. Therefore, all LDS loads are recurrent loads. We also include a small computation loop between each node access. This allows us to evaluate the effects of various ratios between computation and memory access. To eliminate branch effects, we use a perfect branch predictor for these microbenchmark simulations.

```
for (i = 0; i < 10; i++)
  node = head;
  while (node)
    for (j = 0; j < iters; j++)
      sum = sum+j;
    node = node->next;
  endwhile
endfor
```

Figure 6: Microbenchmark Source Code

Figure 7 shows the speedup for each prefetch model over the base system versus compute iterations on the x-axis. The thick solid line indicates the speedup of a perfect memory system where all references hit in the L1 cache. In general, the results for the push model (thin solid line) match our intuition, large speedups (over 100%) are achieved for low computation to memory access ratios, and the benefits decreases as the amount of computation increases relative to memory accesses. However, the increase in computation is not the only cause for decreased benefits: cache pollution is also a factor.

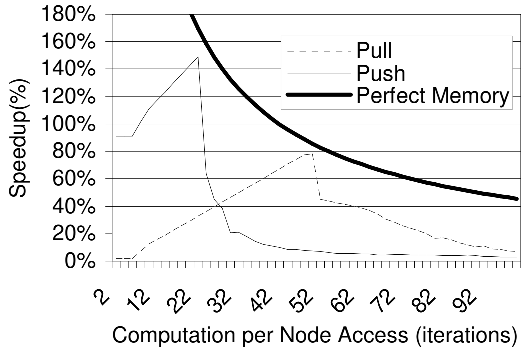


Figure 7: Push vs. Pull: Microbenchmark Results

The dashed line in Figure 7 represents the pull model. We see the same shape as the push model, but at a different scale. It takes much longer than the push model for the pull model to reach its peak benefit. This is a direct consquence of the serialization caused by pointer dereferences; the next node address is not available until the current node is fetched from the lower levels of the memory hierarchy.

We consider three distinct phases in Figure 7. Initially, the push model achieves nearly 100% speedup. While this is not close to the maximum achievable, it is significant. These speedups are a result of cache misses merging with the prefetch requests (see Figure 8). In contrast, the pull model does not exhibit any speedup for low computation per node. Although Figure 8 shows that 100% of the prefetch requests merge with cache misses, the prefetch is issued too late to produce any benefit.

As the computation per node access increases, the prefetch engine is able to move further ahead in the list traversal, eventually eliminating many cache misses. This is shown in Figure 8 by the increase in prefetch buffer hits and decrease in miss ratio. Although the miss ratio and prefetch merges plateau, performance continues to increase due to the prefetch merges hiding more and more of the overall memory latency. Eventually, there is enough computation that prefetch requests no longer merge, instead they result in prefetch hits. This is shown by the spike in prefetch hits (the dashed line) in Figure 8, and occurs at the same iteration count as the maximum speedup shown in Figure 7.

Further increases in computation cause the prefetch engine to run too far ahead, thus causing pollution. As new cache blocks are brought closer to the processor, they replace older blocks. For sufficiently long delays between node accesses, the prefetched blocks begin replacing other prefetched blocks not yet accessed by the CPU. This conclusion is supported by the decrease in prefetch buffer hits shown in Figure 8 and an increase in the L2 miss ratio shown in Figure 9.

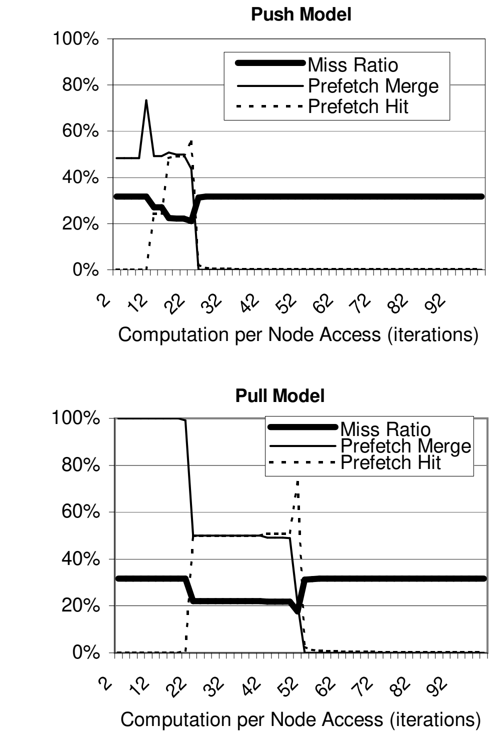


Figure 8: Microbenchmark Prefetch Performance

Although the L1 miss ratio returns to its base value, we continue to see significant speedups. These performance gains are due to prefetch benefits in the L2 cache. Recall, that we place prefetched cache block into both the L2 and prefetch buffer. Even though the prefetch engine is so far ahead that it pollutes the prefetch buffer, the L2 still provides benefits. Figure 9 shows that the global L2 miss ratio, remains below the base system. However, the miss ratio increases, hence the speedup decreases, as the computation per node increases because of L2 cache pollution.

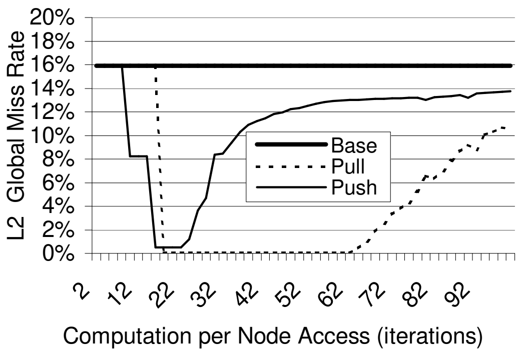


Figure 9: Push vs. Pull: Microbenchmark L2 Cache Global Miss Ratio

The results obtained from this microbenchmark clearly show some of the potential benefits for the push model. However, there are some opportunities for further improvement, such as developing a technique to "throttle" the prefetch engine to match the rate of data consumption by the processor, thus avoiding pollution. We leave investigation of this as future work.

4.3 Macrobenchmark Results

While the above microbenchmark results provide some insight into the performance of the push and pull models, it is difficult to extend those results to real applications. Variations in the type of computation and its execution on dynamically scheduled processors can make it very difficult to precisely predict memory latency contributions to overall execution time.

To address this issue, we use four list-based macrobenchmarks: em3d, health, and mst from the Olden pointer-intensive benchmark suite [18] and Rayshade [9]. Em3d simulates electro-magnetic fields. It constructs a bipartite graph and processes lists of interacting nodes. Health implements a hospital check in/out system. The majority of cache misses come from traversing several long link lists. Mst finds the minimum spanning tree of a graph. The main data structure is a hash table. Lists are used to implement the buckets in a hash table. Rayshade is real-world graphics application that implements a raytracing algorithm. Rayshade divides space into grids and objects in each grid are stored in a linked-list. If a ray intersects with a grid, the entire object list in that grid is traversed.

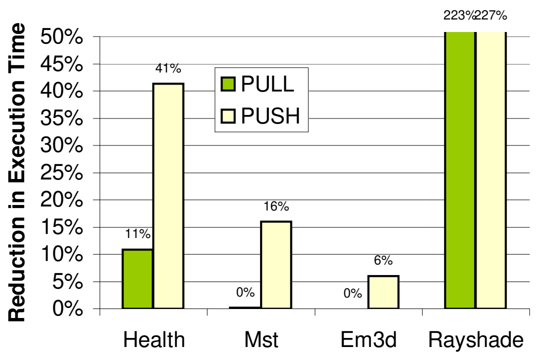


Figure 10: Push vs. Pull: Macrobenchmark Results

Figure 10 shows the speedup for both push and pull prefetching using our benchmarks on systems that incur a one cycle overhead for address translation with a perfect TLB. We examine the effects of TLB misses later in this section. From Figure 10 we see that the push model outperforms the pull model for all four benchmarks. Rayshade shows the largest benefits, with speedups of 227% for the push model, and 223% for the pull model. Health achieves 41% improvement in execution time for the push model while achieving only 11% for the pull model. Similarly, for the push model, em3d and mst achieve 6% and 16%, respectively, while neither shows any benefit using the pull model.

We note that the results for our pull model are not directly comparable to previous pull-based prefetching techniques, since we do not implement the same mechanisms or use the exact same memory system parameters. Recall, that we execute a simple traversal kernel on the L1 prefetch engine to issue prefetch requests. This kernel executes independent of the CPU, and does not synchronize in any way. Most previous techniques are "throttled" by the CPU execution, since they prefetch a specific number of iterations ahead. Furthermore, our kernel may prefetch incorrect data. For example, this can occur in mst since the linked list is searched for a matching key and the real execution may not traverse the entire list. Similarly, in em3d each node in the graph contains a variable size array of pointers to/from other nodes in the graph. Our kernel assumes the maximum array size, and hence may try to interpret non-pointer data as pointers and issue prefetch requests for this incorrect data.

To gain more insight into the behavior of the push architecture, we analyze prefetch coverage, pollution, and prefetch request distribution at different levels of the memory hierarchy. Prefetch coverage measures the fraction of would-be read misses serviced by the prefetch mechanism. To determine the impact of cache pollution, we count the number of prefetched blocks replaced from the prefetch buffer before they are ever referenced by the CPU. Finally, the request distribution indicates where in the memory hierarchy prefetch requests are initiated.

Prefetch Coverage

We examine prefetch coverage by categorizing successful prefetch requests according to where in the memory hierarchy the would-be cache miss "meets" the prefetched data. The three categories are partially hidden, hidden L1, and hidden L2. Partially hidden misses are those would-be cache misses that merge with a prefetch request somewhere in the memory hierarchy. Hidden L1 misses are those L1 misses now satisfied by the prefetch buffer. Similarly, hidden L2 misses are those references that were L2 misses without prefetching, but are now satisfied by prefetched data in the L2 cache. Note, since we deposit data in the L2 cache and the prefetch buffer, blocks replaced from the prefetch buffer can still be resident in the L2 cache.

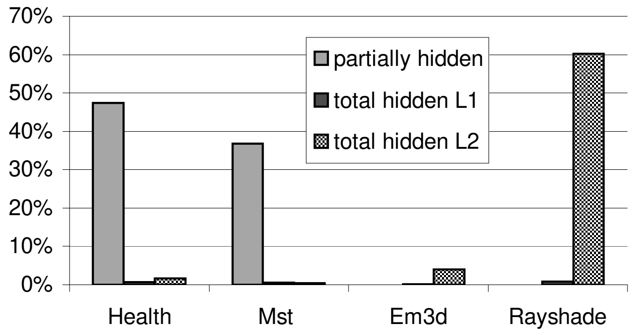


Figure 11: Prefetch Coverage

Figure 11 shows the prefetch coverage for our benchmarks. Our results show that for health and mst, the majority of load misses are only partially hidden, 47% and 35%, respectively. These benchmarks do not have enough computation to overlap with the prefetch latency. Rayshade is the other extreme, with no partially hidden misses, but 60% of L2 misses are eliminated. Em3d shows very little benefit, with only a small number of L2 misses eliminated.

Prefetch Buffer Pollution

To eliminate all L1 misses, a cache block must arrive in the prefetch buffer before the CPU requires the data. Furthermore, the cache block must remain in the prefetch buffer until the CPU accesses the data. If the prefetch engine runs too far ahead it may replace data before it accessed by the CPU.

Figure 12 shows the percentage of blocks placed into the prefetch buffer that are replaced without the CPU accessing the data for the push model. From this data we see that mst, em3d, and rayshade all replace nearly 100% of the blocks placed in the prefetch buffer. Health suffers very little pollution.

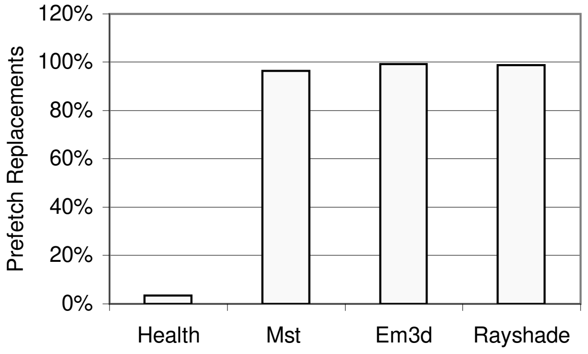


Figure 12: Prefetch Buffer Pollution

These replacements can be caused by the prefetch engine running too far ahead, and replacing blocks before they are used, or by prefetching the wrong cache blocks. Recall, that for mst and em3d our traversal kernel is approximate, and we could be fetching the wrong cache blocks. This appears to be the case, since neither of these benchmarks showed many completely hidden misses (see Figure 11). In contrast, rayshade shows a large number of hidden L2 misses. Therefore, we believe the replacements it incurs are caused by the prefetch engine running too far ahead, but not so far ahead as to pollute the L2 cache.

Prefetch Request Distribution

The other important attribute of the push model is the prefetch request distribution at different levels of the memory hierarchy. The light bar in Figure 13 shows the percentage of prefetch operations issued at each level of the memory hierarchy and the dark bar shows the percentage of redundant prefetch operations. A redundant prefetch requests a cache block that already resides in a higher memory hierarchy level than the level that issued the prefetch.

From Figure 13 we see that health and rayshade issue most prefetches at the main memory level. Em3d issues half from the L2 cache and half from main memory. Finally, mst issues over 30% of its prefetches from the L1 cache, 20% from the L2 and 60% from main memory. Health and rayshade traverse the entire list each time it is accessed. Their lists are large enough that capacity misses likely dominate. In contrast, mst performs a hash lookup and may stop before accessing each node, hence conflict misses can be a large component of all misses. Furthermore, the hash

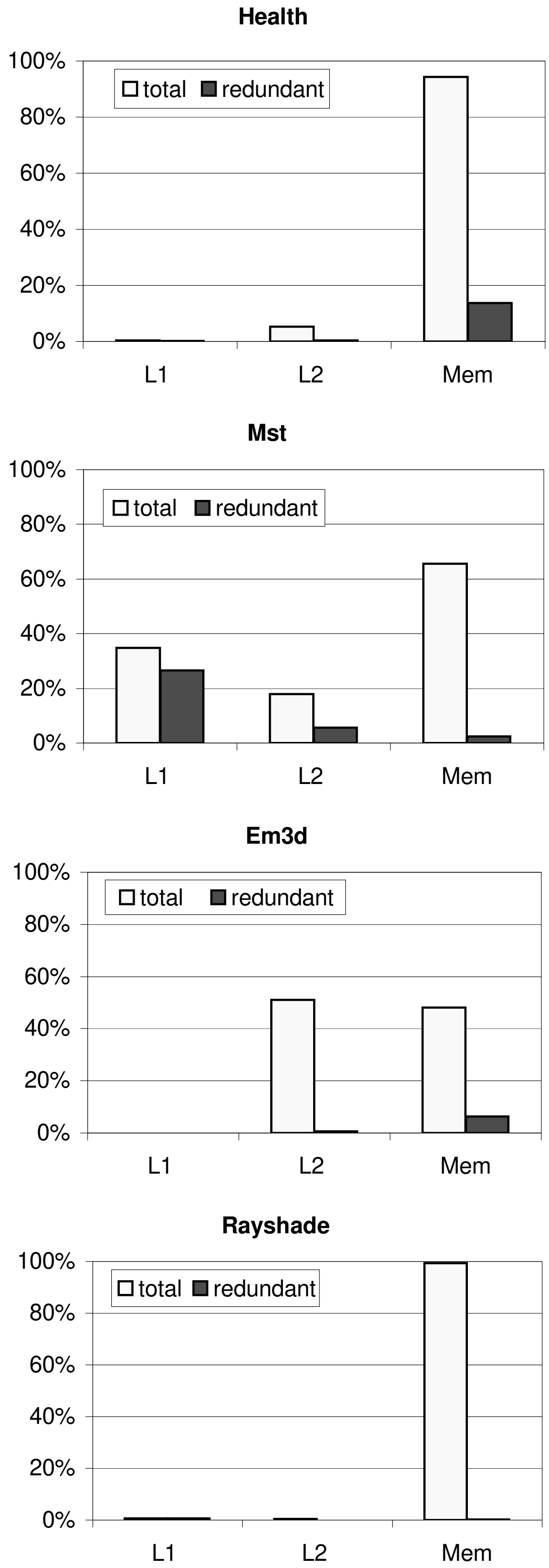


Figure 13: Prefetch Request Distribution

lookup is data dependent and blocks may not be replaced between hash lookups. In this case, it is possible for a prefetch traversal to find the corresponding data in any level of the cache hierarchy, as shown in Figure 13.

Mst also exhibits a significant number of redundant prefetch operations. Most of the L1 prefetches are redundant, while very few of the memory level prefetches are redundant. For health and em3d, 15% and 7%, respectively, of all prefetches are redundant and issued from the memory level. Rayshade incurs minimal redundant prefetch requests. Redundant prefetches can prevent the prefetch engine from running ahead of the CPU and they compete with the CPU for cache, memory and bus cycles. We are currently investigating techniques to reduce the number of redundant prefetch operations.

Impact of Address Translation

Our proposed implementation places a TLB with each prefetch engine. However, the results presented thus far assume a perfect TLB. To model overheads for TLB misses, we assume hardware managed TLBs with a 30 cycle miss penalty [19]. When a TLB miss occurs at any level, all the TLBs are updated by the hardware miss handler.

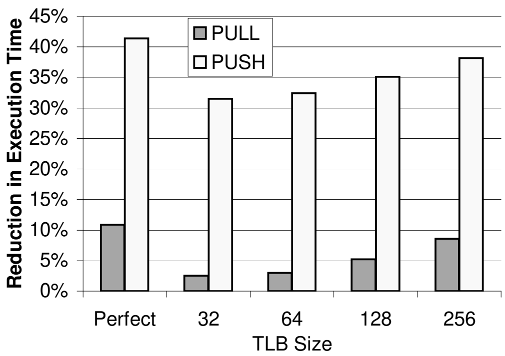


Figure 14: Impact of Address Translation for Health

For most of our benchmarks, the TLB miss ratios are extremely low, and hence TLB misses do not contribute significantly to execution time. Health is the only benchmark with a noticeable TLB miss ratio (17% for a 32-entry fully-associative TLB). Therefore, we present results for only this benchmark.

Figure 14 shows the speedup achieved for various TLB sizes, including a perfect TLB that never misses. From this data we see that smaller TLB sizes reduce the effectiveness of both push and pull prefetching. However, we see that the push model still achieves over 30% speedup even for a 32-entry TLB, while the pull model achieves less than 5%. Increasing the TLB size, increases the benefits for both push and pull prefetching.

5. RELATED WORK

The early data prefetching research, both in hardware and software, focused on regular applications. On the hardware side, Jouppi proposed to use stream buffers to prefetch sequential streams of cache lines based on cache misses [7]. The original stream buffer design can only detect unit stride. Palacharla et. al [15] extend this mechanism to detect non-unit stride off-chip. Baer et. al [2] proposed to use a reference prediction table (RPT) to detect the stride associated with load/store instructions. The RPT is accessed ahead of the regular program counter by a look-ahead program counter. When the look-ahead program counter finds a matching stride entry in the table, it issues a prefetch. The Tango system [16] extends Baer's scheme to issue prefetches more effectively on 1 processor. On the software side, Mowry et. al [13] proposed a compiler algorithm to insert prefetch instructions in scientific computing code. They improve upon the previous algorithms [5; 8; 17] by performing locality analysis to avoid unnecessary prefetches.

Correlation-based prefetching is another class of hardware prefetching mechanisms. It uses the current state of the address stream to predict future references. Joseph et. al [6] build a Markov model to approximate the miss reference string using a transition frequency. This model is realized in hardware as a prediction table where each entry contains an index address and four prioritized predictions. Alexander et. al [1] also implement a table-based prediction scheme similar to [6]. However, it is implemented at the main memory level to prefetch data from the DRAM array to SRAM prefetch buffers on the DRAM chip. Correlation-based prefetching is able to capture complex access patterns but it has three shortcomings: (i) it cannot predict future references as far ahead of the program execution as the push architecture; (ii) the prefetch prediction accuracy decreases if applications have dynamically changing data structures or LDS traversal orders, and (iii) to achieve good prediction accuracy, it requires the prediction table proportional to the working set size.

Another class of data prefetching mechanisms focuses specifically on pointer-intensive applications. Data structure information is used to construct solutions. The Spaid scheme [10] proposed by Lipasti et. al is a compiler-based pointer prefetching mechanism. It inserts prefetch instructions for the objects pointed by pointer arguments at call sites. Luk et. al [11] proposed three compiler-based prefetching algorithms. Greedy prefetching schedules prefetch instructions for all recurrent loads a single instance ahead. History-pointer prefetching modifies source code to create pointers to provide direct access to non-adjacent nodes based on previous traversals. This approach is only effective for applications that have static structures and traversal order. This method incurs both space and execution overhead. Data-linearization prefetching maps heap-allocated nodes that are likely to be accessed close together in time into contiguous memory locations so the future access addresses can be correctly predicted if the creation order is the same as the traversal order.

Chilimbi et. al [4] present another class of software approaches to pointer-intensive applications. They proposed to perform cache conscious data placement considering several cache parameters such as the cache block size and associativity. The reorganization process happens at run time. This approach greatly improves the data spatial and temporal locality. The shortcoming of this approach is that it can incur high overhead for dynamically changing structures, and it cannot hide latency from capacity misses.

Zhang et. al [23] proposed a hardware prefetching scheme for irregular applications in shared-memory multiprocessors. This mechanism uses object information to guide prefetching. Programs are annotated to bind together groups of data (e.g., fields in a record or two records linked by a pointer) either by programmers or the compiler. Groups of data are then prefetched under hardware control. The constraint of this scheme is that accessed fields in a record need to be contiguous in memory. Mehrotra et. al [12] extends stride detection schemes to capture both linear and recurrent access patterns.

Recently, Roth et. al [19] proposed a dependence based prefetching mechanism (DBP) that dynamically captures LDS traversal kernels and issues prefetches. In their scheme, they only prefetch a single instance ahead of a given load and wavefront tree prefetching is performed regardless of the tree traversal order. The proposed push architecture allows the prefetching engine to run far ahead of the processor.

In their subsequent work [20], Roth et. al proposed a jump-pointer prefetching framework similar to the history-pointer prefetching in [11]. They provide three implementations: software-only, hardware-only and a cooperative software/hardware technique. The performance of these 3 implementations varies across the benchmarks they tested. For few of the benchmarks which have static structure, traversal order, and multiple passes to LDS, the jump-pointer prefetching can achieve high speedup over the DBP mechanism. However, for applications that have dynamically changing behavior, the jump-pointer prefetching is not effective and sometimes can even degrade performance. The proposed push architecture can give more stable speedup over the DBP mechanism and adjust the prefetching strategy according to program attributes.

6. CONCLUSION

This paper presented a novel memory architecture designed specifically for linked data structures. This new architecture employs a new data movement model for linked data structures by pushing cache blocks up the memory hierarchy. The push model performs pointer dereferences at various levels in the memory hierarchy. This decouples the pointer dereference (obtaining the next node address) from the transfer of the current node up to the processor, and allows implementations to pipeline these two operations.

Using a set of four macrobenchmarks, our simulations show that the push model can improve execution time by 6% to 227%, and outperforms a conventional pull-based prefetch model that achieves 0% to 223% speedups. Our results using a microbenchmark show that for programs with little computation between node accesses, the push model significantly outperforms a conventional pull-based prefetch model. For programs with larger amounts of computation, the pull model is better than the push model.

We are currently investigating several extensions to this work. First, is a technique to "throttle" the push prefetch engine, thus preventing it from running too far ahead of the actual computation. We are also investigating techniques to eliminate redundant prefetch operations, and the use of a truly programmable prefetch engine that may allow us to eliminate incorrect prefetch requests (e.g., em3d). Finally, we are investigating support for more sophisticated pointer-based data structures, such as trees and graphs.

As the impact memory system performance on overall program performance increases, new techniques to help tolerate memory latency become ever more critical. This is particularly true for programs that utilize pointer-based data structures with little computation between node accesses, since pointer dereferences can serialize memory accesses. For these programs, the push model described in this paper can dramatically improve performance.

7. REFERENCES

[1] T. Alexander and G. Kedem. Distributed predictive cache design for high performance memory system. In *Proceedings of the 2th International Symposium on High-Performance Computer Architectur*, February 1996.

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

[3] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors-the simplescalar tool set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[4] T. Chilimbi, J. Larus, and M. Hill. Improving pointer-based codes through cache-concious data placement. Technical Report CSL-TR-98-1365, University of Wisconsin, Madison., March 1998.

[5] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-detected data prefething in multiprocessor with memory hierarchy. In *Prceedindings of International Conference on Supercomputing*, 1990.

[6] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[7] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[8] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, 1991.

[9] C. Kolb. The rayshade user's guide. In *http://graphics.stanford.EDU/ cek/rayshade.*

[10] M. H. Lipasti, W. Schmidt, S. R. Kunkel, and R. Roediger. Spaid: Software prefeteching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, June 1995.

[11] C.-K. Luk and T. Mowry. Compiler based prefetching for recursive data structure. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 222–233, October 1996.

[12] S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric program. In *Proceedings of the 10th International Conference on Supercomputing*, pages 133–139, May 1996.

[13] T. 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 System*, pages 62–73, October 1992.

[14] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[15] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[16] S. Pinter and A. Yoaz. A hardware-based data prefetching technique for superscalar processor. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 214–225, December 1996.

[17] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, 1989.

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

[19] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 115–126, October 1998.

[20] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[21] A. Smith. Cache memories. *Computing Surveys*, 14(4):473–530, September 1982.

[22] G. Sohi. Instruction issue logic for high performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[23] 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.