Dependence Based Prefetching for Linked Data Structures

Amir Roth, Andreas Moshovos and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin, Madison
1210 W. Dayton St.
Madison, WI 53706
{amir, moshovos, sohi}@cs.wisc.edu

Abstract

We introduce a dynamic scheme that captures the access patterns of linked data structures and can be used to predict future accesses with high accuracy. Our technique exploits the dependence relationships that exist between loads that produce addresses and loads that consume these addresses. By identifying producer-consumer pairs, we construct a compact internal representation for the associated structure and its traversal. To achieve a prefetching effect, a small prefetch engine speculatively traverses this representation ahead of the executing program. Dependence-based prefetching achieves speedups of up to 25% on a suite of pointer-intensive programs.

1 Introduction

Linked data structures (LDS) such as lists and trees are used in many important applications. The importance of LDS is growing with the increasing popularity of C++, Java, and other systems that use linked object graphs and function tables. Flexible, dynamic construction allows linked structures to grow large and difficult to cache. At the same time, LDS are traversed in a way that prevents individual accesses from being overlapped. These factors magnify the negative performance impact of off-chip data access.

Prefetching can be an important tool in boosting the performance of applications that use LDS. Historically, however, prefetch mechanisms have had trouble with these structures. Not only do the overlap restrictions reduce the effectiveness with which memory latency can be hidden, but LDS accesses have defied traditional address prediction techniques that drive prefetching activity. These techniques rely on address stream regularities to extract arithmetic patterns that can be used to make predictions. Such patterns are not necessarily found in LDS access sequences. In this work, we propose a new solution that attacks both problems by exploiting dependence information.

We say that two instructions are *dependent* if one produces a value the other consumes, or affects its execution in some other way. Techniques that exploit dependences base analysis and speculation on this relationship, rather than the actual values exchanged. To date, most microarchitectural techniques have used *value-based* speculation techniques. Caches exploit temporal and spatial locality in the set of addresses referenced by the program, branches are predicted using outcomes of previous branches, and values are speculated using histories of previous instruction results. However, recent work [15][16][23][6] has demonstrated that dependence relationships exhibit regularities that can be exploited in ways that the values they exchange cannot. These studies have focused primarily on *memory dependences* that exist between stores and loads that access the same location. Our technique uses *load value dependences*, a class of dependences between loads that produce (load from memory) addresses and those that subsequently consume (access data at) those addresses. Load value dependences capture regularities in the address generation process rather than in the addresses themselves.

Dependence-based prefetching dynamically identifies loads that access linked data structures. It collects these loads along with the dependence relationships that connect them and constructs a description of the steps the program has followed to traverse the structure. Predicting that the program will continue to follow these same steps, a small prefetch engine takes this description and speculatively executes it in parallel with the original program. Since it executes only the loads that are required to touch the data structure's elements, this engine initiates LDS accesses at a rate dictated only by the (memory) latency of each operation. Since the processor executes all instructions, the prefetch engine may run ahead, producing the desired prefetching effect.

The rest of the work is organized as follows. We begin with a discussion of the issues involved in prefetching linked data structures in section 2. In section 3, we briefly introduce our benchmark suite, and present statistics that motivate our solution for this problem. A detailed description of our mechanism is presented in section 4, followed by a quantitative evaluation in section 5. We relate our solution to other work in section 6, then offer our conclusions.

2 Prefetching Linked Data Structures

Linked data structures (LDS) are widely used in compilers, databases, and graphics applications. LDS are constructed by connecting data elements to one another explicitly; elements in an LDS contain fields that name all adjacent elements by address. This mode of connectivity allows the easy construction and manipulation of data structures of arbitrary shape, such as trees and graphs. Dynamic construction also allows LDS to grow very large, making them difficult to cache. Added to this is the fact that accesses to successive LDS elements, and to the data they contain cannot be overlapped, as the process of address generation itself requires an inherently serial evaluation through memory. Commonly known as the *pointer-chasing problem*, this condition effectively exposes the full latency of each LDS access. The key to hiding this latency is to issue LDS accesses as early as possible, overlapping them with other work.

Prefetching can be implemented in both hardware and software. Software schemes [17][12][10] have potentially larger analysis scope and add no complexity to the processor. However, we choose to investigate hardware schemes for several reasons. Hardware mechanisms require no *a priori* program information or transformations, as well as no architectural interface changes. They impose no explicit execution overhead. Hardware techniques have at their disposal the execution profile of the program, as well as other information, like the addresses of LDS elements, that is available only at run-time. Dynamic solutions also have the potential for adapting to program phases, changing conditions in the processor and memory system, and behavior dictated by the input. Finally, a hardware scheme may be able to initiate action earlier than a program supplied cue, since the latter must be "seen" by the processor. This can be a useful property when contending with serialized latencies caused by pointer chasing.

Hardware prefetchers proposed to date [9][1] analyze the address history associated with an instruction or group of instructions. They exploit regularity in the stream to compress the access sequence, quickly regenerating it to produce prefetching addresses. For example, address sequences that exhibit *arithmetic regularity*, such as the ones corresponding to sequential array traversal, can be compressed to a pair of numbers: a base value and a stride. Not only is this representation extremely compact, it has a nice property that allows it to be used as a formula to generate previously unseen addresses that closely match actual program accesses.

In line with these methods, we may attempt to compress LDS access sequences. Ordinarily, a prefetch address for an LDS element cannot be generated until the addresses of all *previous* elements in the structure are known. Compression is attractive because it allows for generation of prefetch addresses for arbitrary LDS elements without the need for a serial evaluation. However, compressing an LDS access stream can be a difficult task. Addresses of adjacent LDS elements are not required to have a regular arithmetic relationship. Linear layout in an LDS is usually the result of allocator strategy, compacting garbage collection, or careful hand optimization, and is often compromised as the data structure evolves. In the absence of such regularity, we expect the size of the compressed form to be proportional (smaller but certainly not constant) to the size of the LDS itself. This property potentially makes compression of large structures inconvenient. Even in the event that sufficient compression is possible, it is unlikely that the compressed format could be used to generate previously unseen addresses.

To handle the case in which address regularities are not available and compression is not possible, we make the observation that the instructions used by the program to access a particular set of LDS elements, are themselves a compact formula for generating the addresses of those elements. The mechanism we present captures the process of address generation itself and predicts addresses by mimicking this process. In addition, by creating a separate, dependence-based representation for this important kernel of the program, our technique can issue requests for LDS elements with little overhead, and with no interference from other parts of the program. The details of the mechanism are described in section 4. As motivation, we first present a brief analysis of LDS access behavior in a suite of programs.

3 A Study of Pointer Intensive Programs

The technique we propose improves performance by hiding memory latency associated with LDS access. Its effectiveness will be a function of three factors: (i) the number of LDS accesses in the program and their contribution to the total latency associated with the memory system, (ii) the amount of work in the program that can be overlapped with this latency, and (iii) our mechanism's ability to capture this behavior and leverage the available work. In this section, we attempt to quantify the first two parameters by presenting a characterization of LDS access behavior for programs from the Olden pointer-intensive benchmark suite [20]. The Olden benchmarks are a collection of programs that includes small and medium sized scientific codes (*bh* and *em3d*), process simulations (*health* and *power*), graph optimization routines (*mst* and *tsp*), graphics utilities (*perimeter* and *voronoi*), a sorting routine (*bisort*) and a toy tree benchmark (*treeadd*). We use this set of programs as it had been previously used to evaluate compiler prefetching algorithms [12]. A summary of the benchmarks, the sizes and types of linked data structures used, input parameters and dynamic instruction counts is shown in Table 1. In order to compress the subsequent figures, we will refer to the benchmarks by only the first three letters of their name (*e.g., bis* for *bisort*).

LDS-specific memory behavior can be summarized by examining the load instructions that access LDS elements, or *pointer loads* in our terminology. A pointer load is a load whose input base address was produced by another load instruction. This definition encompasses LDS accesses, and distinguishes them from stack and array loads, whose addresses are computed arithmetically, and loads that use addresses produced by a means other than an indirection.

The latency associated with pointer loads is difficult to account for in a way that is not highly dependent on a particular processor configuration; we use data cache miss rate as an alternate metric to give a feel for the magnitude of the problem. Also shown in table 1 for each benchmark are the number of loads (as a percentage of all dynamic instructions) and the data miss rate for a 32KB, 2-way, 32B line data cache. Pointer load behavior is summarized under the headings *pointer loads,* which gives the fraction of all loads that are pointer based, and *pointer load contribution*, which gives the percent of all misses caused by pointer loads.

Pointer loads represent a large fraction of all loads in the Olden benchmarks and contribute a disproportionately larger fraction of

Bench	Pointer Data structures	Input Parameters	Data Set Size	Inst Count	Loads	Pointer Loads	Miss Rate	Pointer Load Contribution
bh	octree	4K bodies	720KB	866M	29.1%	16.3%	0.7%	53.0%
bisort	binary tree	250,000 numbers	1535KB	625M	15.4%	49.1%	1.1%	99.4%
em3d	lists	2000 nodes, arity 10	1670KB	60M	23.5%	59.2%	26.6%	81.5%
health	quadtree, lists	5 levels, 500 iters	925KB	169M	36.2%	81.1%	17.3%	98.3%
mst	array of lists	1024 nodes	20KB	256M	14.6%	41.3%	6.2%	83.5%
perimeter	quadtree	4K x 4K image	6445KB	1619M	17.1%	16.2%	2.7%	99.7%
power	multiway tree,lists	10,000 nodes	313KB	791M	18.9%	12.2%	0.2%	91.6%
treeadd	binary tree	1M nodes	12300KB	196M	20.6%	15.8%	1.4%	97.2%
tsp	binary tree, lists	100,000 cities	5120KB	338M	9.4%	74.0%	2.8%	99.8%
voronoi	binary tree	60,000 points	11100KB	333M	14.3%	71.2%	1.1%	41.1%

Table 1. Olden benchmark suite. *Data structures used, input parameters, data set size, dynamic instruction count, loads, pointer loads as a percentage of all loads, data cache miss rate for a 32KB cache and pointer load contribution to the miss rate.*

the cache misses, accounting for nearly *all* misses in many of the programs. With several exceptions, notably *health*, *em3d*, and *mst* most of these programs have good *a priori* data cache behavior. These programs may still benefit from prefetching if the miss latencies are high and enough work exists to overlap with them.

3.1 Pointer-Load Classification

We find it useful to further classify pointer loads into *recurrent, traversal*, and *data* loads. Members of each category have properties that restrict their overlap with different kinds of work, and can therefore be thought of as being closer or further from the program's critical path. Consequently, their importance to the performance of the program, and to our mechanism, varies.

Recurrent loads are a subclass of pointer loads; they produce addresses consumed by future instances of themselves. Recurrent loads are often used as induction variables in loops (e.g., p = p->next in a list or p = p->left in a tree). It is important to note that although our working definition is restricted to *self-recurrent loads*, loads may feed themselves indirectly (e.g., p = p->left->right). Indirect recurrent loads are lumped together with *traversal loads;* a class of loads that produce addresses for pointer loads other than themselves. Data loads are all pointer loads that are neither recurrent loads nor traversal loads; they load data other than addresses.

As an illustration of the definitions, we consider a short piece of code that processes a list of machine instructions, each represented by a pair of linked structures. The loop source and assembly code are shown in figure 1(a) and (b) respectively, the list itself is shown in part (c). Instruction 6, which loads the next field of an element, is a *recurrent load*. Instruction 3 loads the address of the pat structure and is a *traversal load*. Instruction 4 loads the code field and is a *data load*. Also shown in figure 1(c) are three instances of each load corresponding to three loop iterations.

The important aspect of our classification scheme is that it partitions loads according to the type of work that can be used to in overlapping and hiding their latency. We illustrate this using two examples. In the first, we try to hide the latency of a recurrent

(a)

```
for (insn = f; insn; insn = insn->next)
  process(insn->pat->code);
```

(b)

```
         1: lw $24, f             insn = f
         2: beq $24, $0, $Exit
$Head:   3: lw $15, 8($24)        insn->pat
         4: lw $4, 0($15)         insn->pat->code
         5: jal process
$Else:   6: lw $24, 4($24)        insn = insn->next
         7: bne $24,$0,$Head
$Exit::
```

(c)

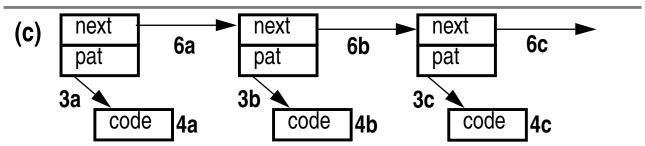

Figure 1. LDS traversal example. *(a) Source and (b) machine code that traverses a linked list. (c) List layout in memory.*

load, load 6, which has a latency longer than the execution time of a single iteration. Since loads cannot be overlapped with loads that depend on them, we see that 6's latency can only be overlapped with work from the same iteration (e.g., 6c with 3c and 4c), leaving the rest exposed. We cannot prefetch 6c effectively because to do so would require that 6b complete execution before its corresponding iteration. Similar restrictions apply to traversal loads (e.g., load 3). In the second case, we attempt to hide the latency of a data load, 4. This can be done if 6 hits in the data cache. Namely, we prefetch 6b as soon as 6a completes, and use the prefetched value to prefetch 3c and then 4c. The latency of 4c is thus overlapped with some work from the previous iteration. These examples suggest that handling recurrent and traversal loads efficiently is the key to prefetching LDS.

3.2 Quantifying Available Work

In the previous sections, we identified the work available for overlapping with pointer loads, especially recurrent and traversal loads, as being important in a prefetching solution. We now quantify this available work. To do so, we measure the distance in dynamic instructions between a pointer load and the *closest* load that produces its base address. Multiple loads may produce the same address, for example when the address is passed as a parameter via the stack. Although choosing the closest load represents the worst case for our mechanism, it provides an unambiguous metric. Figure 2 presents cumulative distributions of these distances for (a) all pointer loads and (b) recurrent loads.

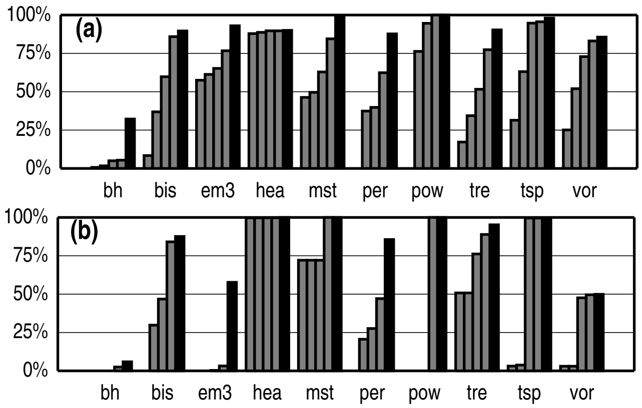

Figure 2. Cumulative address-producer distance distribution. *Distance between a pointer load and the closest producer of its base address. Distances of at most 8, 16, 32, 64, 128 (gray), and 256 (black) dynamic instructions for (a) all pointer loads and (b) recurrent loads.*

The results shown in figure 2 are mixed. Programs like *bh*, *bisort*, *em3d*, *perimeter*, *power*, and *voronoi* contain a large number of recurrent loads with long producer distances (over 128 dynamic instructions). *Health*, *mst*, *treeadd*, and *tsp* have a large representation of short dependence-distance recurrent loads, indicating an abundance of tight loops and a potential lack of work for overlapping with prefetches. However, dependence-based prefetching may still have a positive effect by hiding some of the latency associated with these loads. In addition, these programs have traversal and data loads with somewhat longer producer distances, indicating that prefetching has the opportunity to be successful. We now present a mechanism that attempts to exploit as much of the available work as possible to tolerate pointer load latency.

4 A Dependence-Based Prefetch Mechanism

Dependence-based prefetching dynamically extracts the program kernel responsible for computing addresses of LDS elements. It then speculatively and aggressively executes this kernel alongside the original program. Prefetching is achieved as the engine advances ahead of the main program. In this section, we describe the goals and intended operation of a prefetching mechanism that can predict linked structure access and effectively tolerate serialized latencies. We use these goals to derive a set of requirements for a dependence-based approach. These, in turn, drive our proposed implementation.

We illustrate the desired effect of an LDS prefetching mechanism using the linked list example of the previous section. Figure 3(a) shows an abstract processor executing the program fragment from figure 1. We show the dynamic instruction stream with all instructions currently in the processor's window shaded. Let us assume that a prefetch engine has identified load 1 as producing the value that initiates the load 6 recurrence, and that load 6 has been targeted for prefetch. We would like prefetching to proceed in dataflow fashion. That is, as soon as an instance of load 6 completes, a prefetch for the next instance should be issued immediately. This rapid sequence of prefetches is shown in figure 3(a). The processor uses the value loaded by 1 to fetch the second list element. The prefetch engine takes over from there, prefetching an element as soon as the address of the previous element becomes available. We note that, using this scheme and allowing for some rough timing assumptions, the prefetch for the *fourth* element (6c) may be issued by the prefetch engine before the processor even sees the load corresponding to the *third* element.

The ability to forge ahead of the current instruction window is an important feature that allows a potential solution to attack serialized latencies more efficiently than a typical dynamically-scheduled processor. An out-of-order machine, shown in figure 3(a), can *approximate* the effect of the scheme we present by scheduling pointer loads as soon as their inputs are ready. It may, for example, issue instruction 6 as soon as instruction 1 completes. However, to do so requires that the processor both (i) see instruction 6, and (ii) understand that it is in some way more important than instruction 3 and issue it first. Dependence-based prefetching effectively meets these two requirements by considering only pointer loads. First, it prioritizes recurrent loads. More importantly, it can initiate prefetches for loads that the processor has not seen. Advancing sufficiently far ahead of the processor and opening up enough distance between the prefetch and the target load, allows dependence-based prefetches to cover long LDS access latencies. While this does not constitute a solution to the pointer-chasing problem per se, it does overlap the latency of a given pointer load with all available work starting with the production of its base address.

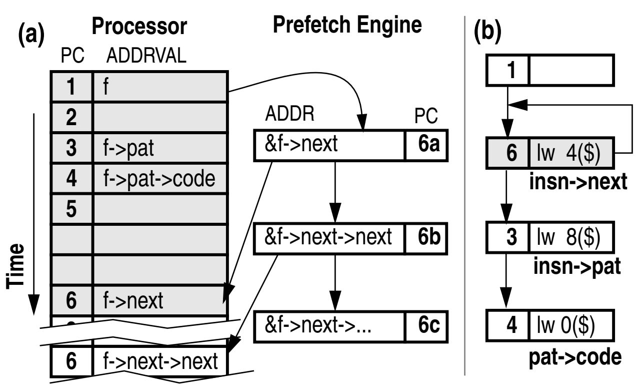

Figure 3. High-level dependence-based prefetching example. *(a) High level description of the prefetch effect we hope to achieve. (b) The abstract internal representation of the list required to drive this mechanism.*

To achieve the effect we described, a mechanism must: (i) identify instructions that participate in traversal (1, 3, 4 and 6 in our example), (ii) activate instances of these instructions with the appropriate input values and (iii) do so as soon as those input values become available. We satisfy these requirements by exploiting the dependence relationship that exists between the loads that produce addresses and those that use them. We use dependence information to rephrase our requirements: as each address is loaded, we predict the loads that will use that address, and issue prefetches for them immediately. It is interesting to note that this process is self-recurrent, as the completed prefetches may themselves be used to launch new prefetches.

We aim to provide structures that make the process of finding potential consumers of a given address simple, and use these to drive the prefetching process. At an abstract level, the information we need to represent can be thought of as a graph. Figure 3(b) shows the graph representation for the list traversal. This graph encodes both the *structural definition* of the list and the *steps* the program took to traverse it. The prefetch schedule in part (a) was generated by "unrolling" the shaded part of this representation.

With a high level understanding of how the dependence-based prefetching functions, we go into a detailed description of several of its important aspects. Section 4.1 describes how information is gathered and used to construct a representation for a particular LDS. In section 4.2, we show how prefetches are requested and serviced by the memory system. Section 4.3 describes how the prefetching process is throttled to minimize erroneous prefetches. In each section, we provide a simple implementation of the corresponding structures and use our running example (insn = insn->next) to demonstrate their function. Finally, we give a short qualitative example in the context of prefetching a binary tree.

4.1 Constructing an LDS Representation

In this section, we describe how LDS traversal is represented using dependences, and how these dependences are identified and captured. To make the rest of the discussion more concrete, we begin with the representation.

The component responsible for storing dependence information is the *Correlation Table* (CT). Each *correlation* represents a dependence between a load instruction that produces an address (PR) and a subsequent load that uses (consumes) that address (CN). In addition to producer and consumer identities (an instruction's identity is its PC), each correlation also contains an *address generation template* (TMPL), which is a condensed form of the consuming load itself. A template contains an opcode and an offset only. A correlation implicitly contains a source identifier (the producer). Destination specifiers are incidental since templates are instantiated for their prefetching effect only. The CT may be implemented as a cache indexed by the producer and should be associative to some degree, as a single producer may feed multiple consumers.

The dynamic creation of correlations requires that we identify loads that produce addresses, identify loads that consume those addresses, and pair producers with consumers even though they might be far apart in the dynamic instruction stream. To do so, the processor maintains a list of the most recently loaded values and the corresponding instructions. This structure, the *Potential Producer Window* (PPW), may be implemented as a queue or a cache containing load value (ADDRVAL) and producer (PR) pairs, indexed by the load value to facilitate matching. Figure 4(a) shows the CT and PPW.

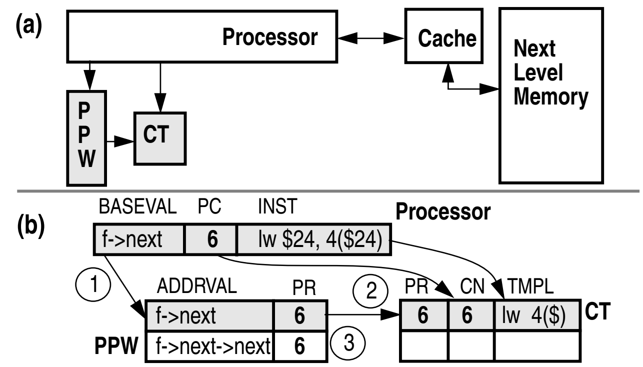

Figure 4. CT and PPW working example. *(a) Block schematic of PPW and CT. (b) The PPW and CT capture the recurrence between instruction 6 and itself (insn = insn->next).*

Correlations are created at instruction commit time. As a load commits, its base address value is checked against entries in the PPW, with a correlation created on a match. The load and its target value are then recorded in the PPW for checking against future loads. This process is illustrated in figure 4(b), which shows how the self dependence of load 6 (insn = insn->next) is captured. As load 6 commits, its base address value (BASEVAL) is looked up in the PPW, which indicates that the previous instance of 6 was the last to load this address (action 1, circled). A new correlation is inserted into the CT establishing the dependence from 6 to itself (action 2). Finally, the value loaded by the current instance of load 6 is entered into the PPW (action 3).

We close this section with two comments regarding the dependence detection process. First, we note that not all loads are potential consumers, nor are all loads producer candidates that must be entered into the PPW. As an optimization, we dismiss loads based off the stack and global pointers as potential consumers since their base addresses are computed via addition. As potential producers, we consider only loads that access address-sized quantities. This is only a heuristic and by no means a substitute for true type information. Many loads that fit the size criteria (e.g., instruction 4 in our running example) do not load addresses. These *false address loads* reduce the effective size of the PPW and contend for CT ports. A further optimization would involve identifying (true) address loads using compiler analysis or profiling, and communicating this information to the processor using a hint.

Finally, we observe that although the prefetch engine is dependence based, dependences are captured using values (addresses). This organization is particularly suitable for our application. Pointer addresses flow from producer to eventual consumer unchanged by arithmetic manipulation. Furthermore, numeric values associated with addresses are rarely seen in other contexts, allowing us to assume safely that two instructions that name the same address are actually related. More importantly, using values allows us to capture dependences accurately, ignoring intermediate register moves and spills to and from memory. Finding earlier producers enables more work to be overlapped with a given miss.

4.2 Prefetch Issue and Use

In this section, we describe how prefetch requests are issued, how they are serviced by the memory system, and how the results are used by subsequent loads. The organization we present is driven by the beliefs that data ports are precious and the prefetch engine should use them only when they are idle, and that prefetched blocks should be kept out of the data cache until they are known to be useful. In line with these requirements, we introduce two new structures. The *Prefetch Request Queue* (PRQ) buffers prefetch requests until data ports are available to service them. The *Prefetch Buffer* (PB) is a small data cache that temporarily holds prefetched blocks. The PRQ and PB are shown in figure 5(a).

Prefetch requests are issued to the PRQ when an address load completes in the processor. A completed load probes the CT in search of potential consumers. On a match, a prefetch address is formed by applying the address generation formula to the value just loaded and a request is enqueued onto the PRQ on behalf of the consumer. Figure 5(b) illustrates this sequence for our running example. An instance of load 6 completes and queries the CT (action 1). Finding the self correlation, it computes the address of the next list element using the loaded value and the correlation formula (action 2). A prefetch request for this address is tagged with the appropriate consumer and enqueued (action 3).

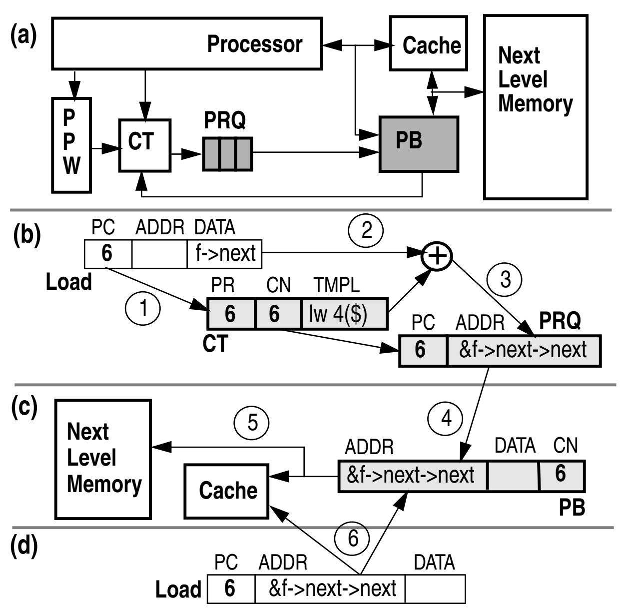

Figure 5. Prefetch example. *(a) Block schematic of the PB and PRQ. (b) A completed load probes the CT, finds a potential consumer and enqueues a prefetch request onto the PRQ. (c) If a data cache port is free, the prefetch request is dequeued and issued to the prefetch buffer. The prefetch buffer checks the first level cache for the block, issuing a request to the second level cache on a miss. (d) A load uses the prefetched block.*

Prefetch requests are dequeued from the PRQ and serviced by the memory system when a data cache port is free. The PB attempts to extract the block from the first level cache, issuing a request to the second level cache on a miss. Spurious requests (e.g., attempting to chase a null pointer or access an unmapped page) are simply dropped. In figure 5(c), the request made by instruction 8 is dequeued and placed into the PB (action 4). Since the corresponding block is not found in the first level cache, a request is issued to the next level (action 5).

Since we would like the prefetch engine to run ahead of the processor, it is important that completed prefetches be themselves able to spawn other prefetches. To facilitate this, the PB maintains a list of requesting consumers (CN) with each block. When a prefetched block arrives, each consumer on the list assumes the role of a producer, probes the CT and potentially generates further requests. An illustration of these steps can be obtained by substituting a completed prefetch for the completed load in figure 5(b).

In figure 5(d), a load instruction picks up a value from the prefetch buffer. The PB and the data cache are accessed in parallel. A cache miss will bring the block into the cache as usual. However, the processor need not wait if the data is available in the PB.

4.3 Simplifying Prefetch Throttle and Control

Allowing the prefetch engine to run arbitrarily far ahead of the processor is undesirable. First, if the prefetch engine gets too far ahead, it may overwrite useful data before the processor has had a chance to use it. We call this phenomenon *early prefetching*. Second, prefetching is speculative, and by definition subject to misspeculation. Should the prefetch engine choose the wrong prefetching path, when traversing a tree for instance, we would like to keep the length of this excursion to a minimum.

Crafting a general solution that would throttle prefetching activity seems complicated. First, we would probably need to keep a running log of prefetches made on behalf of every load so that later program instances do not spawn prefetches that duplicate earlier ones. Second, this mechanism would need to detect discrepancies between per-load access sequences of the processor and those of the prefetch engine, and be able to initiate proper recovery. Fortunately, we have found that for our benchmarks, allowing the prefetch engine to run arbitrarily far ahead is unnecessary. In fact, prefetching a single instance ahead of a given load is sufficient.

To reason about why this might be so, we revisit our list example from figure 1(c), and consider the question of whether a prefetch for 6b, triggered by the completion of 6a, should itself trigger a prefetch for 6c. There are two basic cases to consider here, and the answer for both is no. In the first case, there is enough work starting with 6a to fully overlap with the miss latency of 6b. We therefore assume that there will be enough work to hide the latency of 6c if the prefetch is triggered by the completion of 6b. There is no advantage to triggering the prefetch any earlier. In the second case, there is not enough work and the latency of 6b is only partially hidden. Here, the program instruction and its intended prefetch will complete at the same time, and it should make no difference which one triggers the prefetch for 6c.

Of course, the argument we just gave is not the whole story. It *is* possible for different loop iterations to have different execution latencies, and it is possible to "borrow" work from one iteration for use in another. These situations may arise if the loop contains some conditional code, in which case a recurrent load miss during a short iteration can be hidden using work from a previous, longer iteration. Another possibility is for structure elements to be laid out sequentially and packed two or more to a cache line. Here, the processor would incur a miss followed by one or more hits. Prefetching only a single instance ahead prevents us from exploiting situations like these.

Despite this drawback, single instance prefetching has many advantages, not the least of which is a greatly simplified implementation. Enforcing single-instance prefetching can be done using a counter attached to each prefetch request, and does not require per-instruction prefetching state. Second, it issues a single prefetch request per actual memory reference (allowing each instruction to spawn prefetches for the next two instances will generate two requests for every actual load), a feature that keeps prefetching overhead low and trims the bandwidth requirements of the correlation table and prefetch buffer. Finally, it limits errant prefetch chains to a length of one.

4.4 An Example: Prefetching a Binary Tree

The purpose of this section is to provide a qualitative feel for the operation of dependence based prefetching. Specifically, we examine how dependence-based prefetching handles an in-order binary tree traversal (often used in reduction operations). In-order tree traversal is often implemented recursively (depth first) using two induction variables and three instructions: one fetches the left child, the second restores the address of the current node after the left traversal has finished, and the third fetches the right child using the restored value. These instructions are assembled into four correlations which are shown in figure 6(a): (ll) left feeds left (continue traversal down a left path), (rl) right feeds left (begin traversal down left path), (lr) left feeds right (only at leaf nodes), and (sr) restore feeds right (going back up the tree).

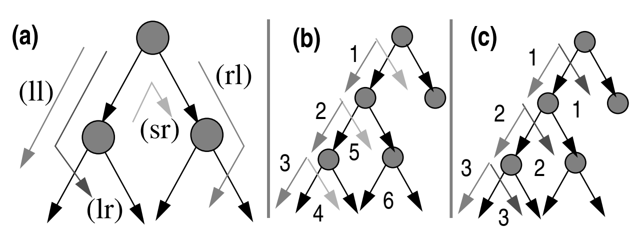

Figure 6. Tree traversal and prefetching. *(a) Four correlations representing tree traversal. (b) Ideal tree prefetching (c) Wavefront tree prefetching performed by our mechanism.*

Traversal, and consequently ideal prefetching, proceeds in the manner shown in figure 6(b), prefetches are shown next to the tree and shaded to match the corresponding correlation. The prefetches in this sequence are issued using only correlations (ll), (rl), and (sr), and prefetch left chains left-to-right and *bottom-up*. Our mechanism issues prefetches using all correlations as shown in figure 6(c). The resulting effect is a left-to-right, *top-down* prefetch order which we call *wavefront*.

Wavefront correctly prefetches down the tree but along the way performs a lot of useless prefetches which correspond to traversal back up the tree. This occurs because the left-feeds-right correlation is assumed to hold at all levels of the tree, even though it is only valid at the leaves. We expect the overall effect of wavefront prefetching to be positive. Near the bottom of the tree, all nodes are likely to fit in the prefetch buffer making order irrelevant. Near the top, wavefront will produce some early prefetches. However, these will not be followed past the first node. Wavefront prefetching should tolerate some latency for at least half the nodes (all the left children), with added benefit near the leaves of the tree. A possible improvement to our scheme that would help in tree prefetching would allow it to *unlearn* or turn off the left-to-right correlation, and eliminate these useless requests. We do not explore such an improvement in this paper.

5 Evaluation

In this section, we provide experimental evidence of the effectiveness of our proposed mechanism. Section 5.1 describes our experimental framework, our benchmarks suite and our simulation environment. In section 5.2, we use execution-driven functional simulation to evaluate our mechanism's ability to correctly predict LDS accesses, measuring prediction accuracy as a function of PPW and CT sizes. We use these to establish an accurate yet reasonable predictor configuration. In section 5.3 we measure the performance impact of dependence based prefetching using detailed timing simulations, and compare the speedups against other, simple prefetching mechanisms. Finally, in sections 5.4 and 5.5, we take a closer look at prefetching itself, and try to gain insight into our performance numbers by measuring its efficiency, overhead, and interaction with the memory system.

5.1 Experimental Framework

Our experiments were performed using the Olden pointer-intensive benchmark suite [20]. The benchmarks were modified by hand to execute on a single processor, and all CM-5 specific code was removed. We compiled the programs for the MIPS-I architecture using the GNU GCC 2.7.2 compiler with optimization flags -O2 and -funroll-loops. Many of the benchmarks contain lengthy, allocation-dominated initialization phases that are not sped up by dependence-based prefetching; we did not optimize or discount these in any way. Finally, the suggested input sets for some benchmarks were changed to produce longer execution samples.

For our simulations, we use the SimpleScalar simulator [2]. We model a 4-way superscalar, out-of-order processor with a conventional five stage pipeline that allows a maximum of 32 in-flight instructions. The branch unit uses a hybrid scheme with an 8K-entry selector table choosing between the outcomes of an 8K-entry, 10 bit history gshare scheme and an 8K-entry 2-bit predictor. Targets are stored in 2K entry, 4-way BTB. The processor has 4 integer ALUs, 4 floating point adders, and single integer and floating point multiply/divide units. ALU operations complete in single cycle, multiply and divide have 3 and 20 cycle latencies. Floating point operations take 2 cycles for addition, 4 for multiplication and 24 for division. The adder is pipelined. The memory system consists of 32KB, 32-byte line, 2-way set-associative first-level instruction and data caches and a 512K, 64-byte line, 4-way set associative shared second level cache. The first level data cache can be accessed in a single cycle, the second level cache latency is 12 cycles to the first word and an additional cycle for each word thereafter. Latency to main memory is 70 cycles. The processor uses 2 read/write ports and a 16 entry load-store queue.

Our prefetching configuration includes a 128 entry PPW, and a 256 entry CT. Prefetch requests wait on a 32 entry PRQ, and are serviced only on cycles when either of the data cache ports is available. We use a 32 entry, 1KB fully associative PB with 4 read/request ports and an access latency of 1 cycle. The PB shares the off-chip data bus with the instruction and data caches; contention on the bus is modeled.

5.2 Address Prediction Accuracy

We measure the ability of our mechanism to capture dependences and use them to predict future LDS addresses. At this point, we are not interested in timing or even the utility of the prefetches themselves. We simply count the fraction of all dynamic pointer loads for which, at the time they were ready to issue, a correlation was present in the CT that both: (i) named the pointer load as the consumer, and (ii) would have produced the correct address. Factors that determine prediction accuracy are the maximum detectable load dependence distance, which prevents the detection and prediction of pointer loads with longer dependences, and the working set size of the correlations themselves. The maximum detectable dependence distance is determined by the size of the PPW, while the correlation working set that can be efficiently represented is given by the number of entries in the CT. This part of the evaluation allows us to estimate the implementation resources that should be devoted to these components in order to achieve reasonable prediction accuracies. Figure 7 shows (a) address prediction accuracy as a function of PPW size given an infinite CT, and (b) as a function of CT size with a fixed 64-entry PPW. We evaluate a fully-associative CT to eliminate aliasing effects.

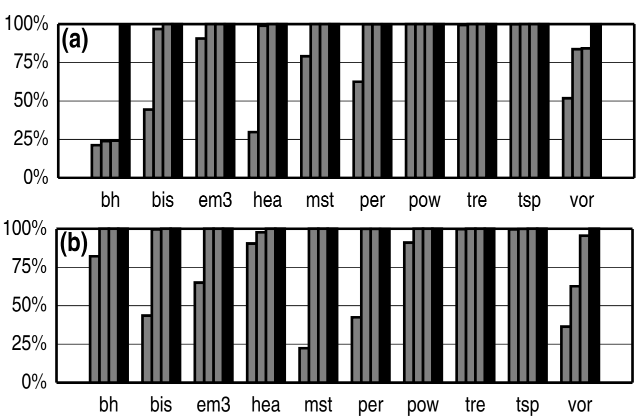

*Figure 7. **Address prediction accuracy.** Percentage of accurately predicted pointer-load addresses. (a) An infinite CT and PPW sizes of 1,4,16, and 64 (black). (b) A 64-entry PPW and CT sizes of 4,16,64 and 256 (black).*

As we claimed earlier, a dependence-based representation has the ability to predict pointer load addresses nearly perfectly. Once the address generation process (producer) for a given pointer load has been identified, addresses for all future instances of the same instruction can be accurately pre-computed. The nearly perfect prediction accuracies we achieve testify to the stability of the dependence relationships. The relatively small structures required to achieve high accuracy, 64 PPW entries and 256 correlations, implies that the correlation working set is small.

5.3 Speedups

We now measure the performance impact of dependence based prefetching. The base machine for the experiment is described in section 5.1. We implement two flavors of the dependence-based prefetching scheme. The first is the one we have been describing all along. The second is augmented with a coarse confidence mechanism that turns off prefetches if the corresponding static load has hit in the first level data cache 8 or more times in a row. These speedups are shown in as light and dark gray bars, respectively in figure 8. We compare these speedups against a naive form of prefetching, namely a system that has twice the on-chip data cache and uses 64, rather than 32, byte lines. Speedups associated with this double data cache configuration are shown in black.

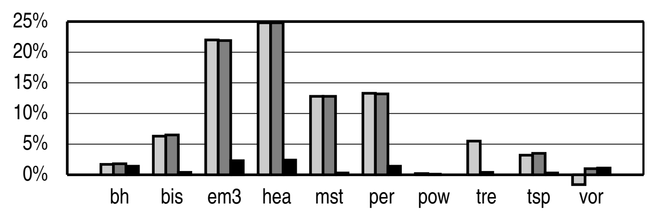

***Figure 8. Performance impact of dependence-based prefetching.** Speedups of dependence based prefetching without (lt gray) and with (dk gray) a coarse confidence scheme, compared to a system that prefetches by doubling the line size, and thus overall size, of the data cache (black).*

Dependence-based prefetching improves the performance of several benchmarks significantly, while having a slight negative performance impact in only one case, *voronoi*. The average speedup for a 1KB prefetch buffer is 10%, significantly outperforming an extra 32KB of data cache. More significant speedups are obtained for *health*, *em3d*, *mst*, and *perimeter.*

Em3d, health, and *mst* are list-based programs with relatively poor cache behavior. Dependence-based prefetching easily captures list traversal behavior and overlaps the element access latencies with the available work. Performance improvement for these benchmarks is roughly proportional to the amount of work in a single loop iteration. *Mst*'s lists are used to implement buckets in a hash table and the loops that traverse them are tight and unable to hide much latency. Performance improvement in *mst* is due to many partially hidden misses. Each iteration of *em3d*'s main loop contains a smaller loop of dependent floating point loads (data pointer loads). This work in each iteration is sufficient to hide the latency of the recurrent loop induction access, and additional benefit is gained by prefetching the floating point data attached to each node. The outer loop in *health* contains quite a bit of computation, but it is the tight inner loops that are responsible for the majority of misses. The benefit we see in this program is due to the terrible *a priori* miss rate and a high dose of partially covered latencies.

Perimeter uses a quadtree and benefits from the wavefront prefetching effect explained in section 4.4. *Bisort*, *treeadd* and *tsp* use binary trees as their primary data structure, and also benefit from the same effect. *Perimeter* sees a larger improvement than the others because more work is available for overlapping at each recursive step. *Treeadd* has so little work at each recursive step, in fact, that the only benefit comes from the wavefront effect near the leaves of the tree. When following the correct traversal, the processor is issuing requests as fast as the prefetch engine. *Bh* and *power* are multiway-tree based programs, but both start out with extremely good cache behavior.

Voronoi uses pointers, but most of its most of its cache misses are caused by array and scalar loads. Most prefetches issued during execution are useless and, combined with a low initial miss rate, contribute little other than bus contention. The resulting 2% slowdown prompted our experiment with the confidence mechanism. The addition of confidence eliminates these unnecessary prefetches and lifts our impact on *voronoi* back into the positive range. However, it also eliminates most of the useful prefetches on *treeadd*, cutting our gains on that benchmark. Experimentation with more elaborate confidence mechanisms is warranted, but is outside the scope of this work.

5.4 A Closer Look at Prefetching

In this section, we attempt to gain some insights into the performance of dependence-based prefetching by taking a closer look at prefetching activity. We begin by presenting a breakdown of all cache blocks prefetched by our mechanism along two axes: block origin (i.e., level in the memory hierarchy) and block utility. These breakdowns are shown per benchmark in figure 9. The bar on the left represents blocks that were resident in the first level cache, the one on the right those that were fetched from the second level cache and potentially main memory. The bottom, darker, portion of each bar represents the fraction of blocks that were used. The combined heights of the two bars add up to 100%, but we split them for clarity.

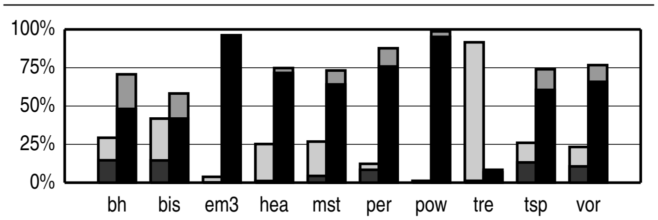

Figure 9. Prefetched block breakdown. *Blocks prefetched from the first level (left) and second level (right) caches. Useful blocks (bottom, dark), and unused blocks (top, light).*

The dark portion of the bars on the right represents the useful work performed by dependence based prefetching. This is the fraction of blocks that were prefetched from the second level cache and used. This category accounts for nearly half of all prefetched blocks in all benchmarks except for *treeadd*, and dominates those benchmarks for which the greatest performance improvement was observed, *em3d*, *health*, *mst*, and *perimeter*. The fact this category is so dominant means that dependence based prefetching is both accurate and efficient. The only application for which this block distribution does not hold true is *treeadd*, which has very little work at each recursive step. The result is that, except near the bottom of the tree, the prefetch engine can only repeat the work of the processor, it cannot prefetch ahead.

The left bar in each series represents the prefetching overhead in some sense. These are the prefetched blocks that were found in the first level cache and copied into the prefetch buffer. These blocks are not entirely useless, since once in the prefetch buffer they may spawn other more useful prefetches. Moving cache blocks into the prefetch buffer has two other positive effects which are illustrated by the fact that these blocks are actually used via the buffer. One possibility is that the block may have been subsequently displaced from the first level cache, in which case the prefetch buffer is assuming the role of pointer-load victim buffer. The second possibility is that the prefetch buffer was used because the data cache ports were busy, in which case the prefetch buffer acts as a bandwidth amplifier. We do not separate the contribution of the two effects here.

5.5 Memory System Performance Metrics

From the memory system standpoint, we quantify both the (hopefully) positive aspects and the overhead in the form of additional bandwidth consumed. We begin by measuring the latency tolerated by prefetched blocks. Here, data cache miss rates do not tell the whole story since the latency of many pointer loads, as well as other loads that access on pointer load cache lines, may be partially hidden. Instead, we present two more telling metrics.

Prefetch *coverage* measures the fraction of would-be load misses serviced by the prefetch mechanism. The height of each bar in figure 10(a) is the sum of the percentage of would-be load misses whose latency was fully tolerated by prefetching (dark, bottom portion), and those whose latency was only partially hidden (light, top portion). For each benchmark, the bar on the left represents pointer loads, and the bar on the right all loads. Since the bar on the right samples more loads than the one on the left, we may expect its overall height to be shorter. However, if enough non-pointer loads benefit from prefetching, by virtue of being on the same cache line as a pointer target for instance, then the effectiveness for loads in general will be higher than for pointer loads in particular. As we predicted in section 3.2, the short dependence distances do not provide much work for overlapping, and consequently, many load misses are only partially masked. However, for the benchmarks that showed the greatest speedups, as many as 75% of all would-be load misses saw some latency reduction.

Prefetch coverage is only a histogram; it does not say how much latency was tolerated for each serviced load nor what that latency is in relation to the other loads. For this reason, we also measure reduction average load wait time, which represents the overall improvement in memory system performance. Normalized average load latencies are shown in figure 10(b), again with pointer loads on the left (in gray) and all loads on the right (black). Not

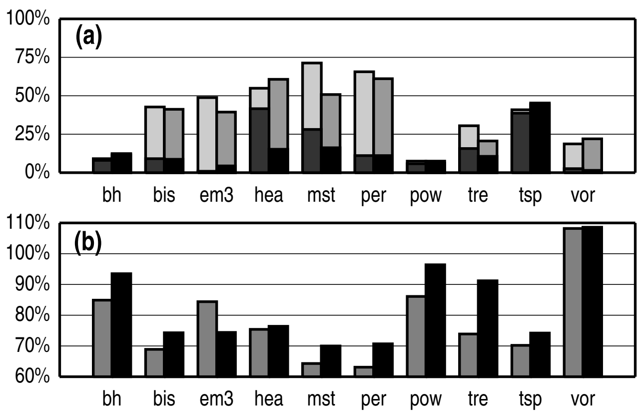

Figure 10. Memory performance improvement metrics. (a) Percentage of would-be load misses serviced by the prefetch buffer. Fully hidden misses (bottom of each bar), partially hidden misses (top), pointer loads (left bar) and all loads (right bar). (b) Normalized average latency for pointer loads (left, gray) and all loads (right, black).

coincidentally, the sharpest improvements correspond to those benchmarks for which dependence based prefetching performs best. For these, the average load wait time was cut by 25%. On several others, *bisort* and *tsp*, a significant decrease in load response time is not translated into a much higher execution efficiency. For these benchmarks, most of the useful prefetches are associated with traversal and data loads that do not execute along the critical path. *Voronoi* is the only program that experiences an increase in load latency.

We quantify the overhead of dependence-based prefetching in terms of increase in the number of accesses to the on chip and second level data caches, as well as to main memory. These increases are shown in figure 11. The dominating overhead, although it is certainly tolerable, is the increased bandwidth demand on the first level data cache ports. This increase, an average of 15% across the benchmarks, is a product of our decision to check prefetch requests for residence in the first level cache, before sending them off-chip. This policy greatly reduces the turn-around time for prefetch requests that are already cache resident, and more importantly, allows dependent useful requests to issue much more quickly. We reiterate that this overhead is not seen by the processor since the ports are used for prefetching only when they are otherwise idle.

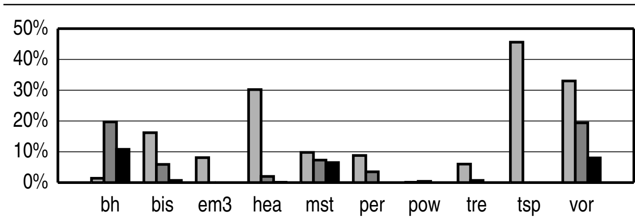

Figure 11. Memory bandwidth overhead. Memory bandwidth usage increases: first level data cache (lt gray), second level cache (dk gray) and main memory (black).

Another benefit of checking blocks for data cache residence before issuing a request off-chip is a substantial reduction in second-level cache bus traffic. The increase we observe in second level cache accesses, an average of 4%, is slight and reinforces our belief that our mechanism is very efficient and accurate. The lack of a more substantial increase means that most prefetches are indeed useful and simply take the place of subsequent reads resulting from would-be first level misses. The 4% increase and the 2% increase in memory bus traffic is due to our mechanism's inability to precisely mimic the traversal of non-linear data structures, such as the ones in *bh* and *voronoi*, and the resulting early prefetches. These figures show that even in the case of serialized latencies, memory bandwidth can be readily traded off for latency.

6 Related Work

Much work has been done in the area of data prefetching, both in software and hardware. Compiler optimizations that improve data locality [13] like blocking and loop interchange can greatly reduce the need for prefetching. However, these fundamentally rely on compile-time knowledge of the data set layout and its interaction with the cache. Linked structures are not often laid out by the compiler, and are incompatible with these optimizations. Software pipelining [10] tolerates high latency loads in loops by increasing the distance between the load and instructions that use its value. While not requiring specific layout information, software pipelining relies on the ability to quickly generate addresses for arbitrary structure elements. LDS access undermines this critical requirement. General purpose software prefetching [17][11] tolerates load latency by scheduling a matching speculative non-faulting load [21] far in advance. Pointer chasing requires that the address for a speculative LDS load be generated using a chain of dependent loads. The critical path of this chain and its relationship to the original load greatly limits the scheduling scope of the prefetch, and consequently, the amount of latency that can be hidden.

Luk and Mowry [12] proposed and evaluated a greedy compiler algorithm for scheduling software prefetches for linked data structures. They showed this scheme to be effective for certain programs, citing instruction overhead and the generation of useless prefetches as performance degradation factors for others. Their algorithm uses type information to identify recurrent pointer accesses, including those accessed via arrays, and may have advantages in tailoring a prefetch schedule to a particular traversal. Our hardware scheme, on the other hand, does not incur instruction overhead, and can prefetch non-pointer data that resides in linked structures. In addition, it provides dynamic detection and suppression of unnecessary prefetches. We expect that this same mechanism can be integrated with a compiler-based prefetch-generation scheme to improve resource consumption.

Luk and Mowry [12] presented a case for history-pointer prefetching, which augments linked structure nodes with prefetching pointer fields, and data-linearization, in which LDS are programmatically laid out at runtime to allow sequential prefetch machinery to capture their traversal. While these schemes have potential for speedup, they also incur serious overheads in the form of runtime storage and additional code needed to maintain history pointers and linear data layout, respectively. Both are difficult to automate.

Another class of software solutions to this problem utilizes cache-conscious data placement [5], the runtime allocation or reorganization of LDS nodes. Clustering techniques pack adjacent LDS nodes into a single (if possible) or consecutive cache lines and improve the spatial locality and arithmetic regularity of LDS access. Coloring techniques eliminate conflicts that occur in common traversals. Data-placement techniques can dramatically improve performance, even when little or no work is available for latency overlapping. However, they incur a potentially high reorganization overhead, making them mostly suitable for relatively static structures. In addition, they are not predictive and do not hide latency resulting from capacity misses. Finally, they require knowledge of the cache parameters. Dependence-based prefetching will mask capacity misses when other work is available, and incurs no explicit overhead.

A similar volume of research has been done in hardware prefetching [3], and dynamic techniques for address prediction [7]. Most of these, such as stream buffers [9], reference prediction table (RPT)[4] and the subsequent Tango [19] analyze address sequences for single instructions arithmetically, and are designed to deal primarily with strided access patterns. Joseph and Grunwald [8] describe Markov predictors which represent cache miss sequences in the form of a probabilistic transition table. Markov predictors are capable of capturing complex patterns, but are nonetheless address based, and require storage proportional to the number of distinct entries in the miss stream.

Mehrotra and Harrison [14] proposed simple extensions to the RPT aimed at capturing recurrent access patterns. They augmented the RPT with a Recurrence Recognition Unit (RRU), a finite state machine able to recognize single level recurrences, such as the ones used in list traversal. The RRU is an efficiently implemented mechanism that leverages structures used for arithmetic prefetching, and captures list access, the most common LDS traversal. Like the RPT, the RRU analyzes address streams on a per-instruction basis, and does not capture dependence between multiple instructions that arise in tree and graph traversals. Dependence-based prefetching can capture and prefetch all pointer loads. However, it has a potentially higher implementation cost.

The use of data dependence between instructions as an information primitive and unit of prediction was introduced by Moshovos, Breach, Vijaykumar and Sohi [15], and later refined by Chrysos and Emer [6]. In the initial work, dependence prediction was used to synchronize loads, avoiding misspeculation due to unresolved dependences. Tyson and Austin [23] and Moshovos and Sohi [16] broadened the scope of use of dependence information. They propose to dynamically and transparently convert address-based activity to dependence-based activity, to reduce memory communication latency. We are not aware of any work that uses instruction dependence speculation to prefetch.

Other related works include the static access/execute decoupling proposed by Smith [22] and subsequent dynamic dependence-based decoupling [18]. Dependence-based prefetching speculatively decouples the LDS traversal portion from the remainder of the program, but does so selectively based on address dependence.

7 Summary and Future Directions

We introduce a dependence based mechanism that dynamically captures and represents pointer access behavior, and uses the representation for prefetching linked data structures (LDS). Dependence-based analysis does not rely on regularities in the address stream, capturing address generation activity explicitly. As a result, it successfully predicts LDS access sequences that exhibit little or no arithmetic patterns. We show that a dependence based mechanism can capture and correctly predict nearly all of the accesses performed by an actual LDS traversal. A prefetch scheme using this mechanism can boost performance of pointer intensive programs by 1% to 25%. We make the following contributions:

(i) We characterize pointer loads and show that, in a suite of pointer-based programs, these are responsible for a significant and often disproportionate fraction of the data cache misses. We categorize pointer loads into data, traversal, and recurrent loads and describe how the latency associated with members of each category may be tolerated.

(ii) We present a new dependence-based mechanism that can correctly predict future LDS accesses by capturing and mimicking the LDS traversal behavior of the executing program. Our scheme is based on the identification of dependence relationships between loads that produce LDS element addresses, and loads that consume them. We show that these dependence relationships are stable and have a small working set, leading to high address prediction accuracies.

(iii) We show that a dependence-based representation enables aggressive, greedy prefetching of linked structures. While not strictly overcoming pointer chasing, this mode of execution can overlap a large fraction of the available work with serialized latencies.

The implementation we propose is a single point in an unexplored design space. Many other designs are possible, for example ones that prefetch directly into the cache. There is potential work in the interpretation of the dependence graphs and prioritization of prefetch operations. The CT may be used to actively classify load instructions according to the number and type of outgoing dependences. This classification scheme can drive prefetching decisions, as well as scheduling policies. In section 4.4, we described the problems associated with tree traversal, and outlined a potential solution involving the dynamic disabling of one dependence. A dynamic implementation of such a mechanism, or an extended version that can prune arbitrary prefetch requests and improve resource contention and PB pollution, is a possibility as is the design of an efficient scheme to allow the prefetch engine to run further ahead.

Future work we find most exciting, however, deals with the exploration of novel microarchitectural techniques enabled by dynamically collected dependence information. Capturing linked data structure access and using it for prefetching is a first step in this direction. Pointer dependences are easy to find since the addresses flow from producer to eventual consumer, unchanged through registers and spills to and from memory. There are other data structures, sparse matrices and index trees for instance, whose traversal does not yield address sequences with arithmetic properties. The nature and organization of mechanisms that can capture and efficiently represent and exploit these access behaviors is an open question. Finally, other uses of dependence information may be possible, in areas unrelated to prefetching in particular or memory system management in general.

Acknowledgments

The authors would like to thank Jim Smith, Mark Hill, Doug Burger, and Milo Martin for their comments on early versions of this paper, and the anonymous referees for their careful reviews and suggestions. Martin Carlisle supplied the Olden benchmarks.

This work was supported in part by NSF grant MIP-9505853, by U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order D346, and by a donation from Intel. The views and conclusions presented are those of the authors and do not necessarily represent the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca or the U.S. Government.

References

[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 Conference on Supercomputing*, pages 176–186, 1991.

[2] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, Jul. 1996.

[3] T. Chen and J. Baer. A performance study of software and hardware prefetching techniques. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, Apr. 1994.

[4] T. Chen and J. Baer. Effective hardware based data prefetching for high performance processors. *IEEE Transactions on Computers*, 44:609–623, May. 1995.

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

[6] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[7] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *Proceedings of the 11th International Conference on Supercomputing*, pages 196–203, Jun. 1997.

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

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

[10] M. Lam. Software pipelining: an efficient scheduling technique for vliw machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 318–328, Jun. 1988.

[11] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. Spaid: Software prefetching in pointer and call intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 231–236, Nov. 1995.

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

[13] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, Jul. 1996.

[14] 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.

[15] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, Jun. 1997.

[16] A. Moshovos and G. Sohi. Streamlining inter-operation communication via data dependence prediction. In *Proceeding of the 30th Annual Internation Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[17] T. Mowry, M. 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*, pages 62–73, Oct. 1992.

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

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

[20] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, Mar. 1995.

[21] A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 32–50, Oct. 1992.

[22] J. Smith. Decoupled access/execute computer architecture. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, Jul. 1982.

[23] G. Tyson and T. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proceeding of the 30th Annual Internation Symposium on Microarchitecture*, pages 218–227, Dec. 1997.