University of Maryland Systems & Computer Architecture Group technical report UMD-SCA-TR-2000-01.

Multi-Chain Prefetching: Exploiting Natural Memory Parallelism in Pointer-Chasing Codes

Nicholas Kohout, Seungryul Choi, and Donald Yeung
Department of Electrical and Computer Engineering and
Institute for Advanced Computer Studies
University of Maryland at College Park

June 20, 2000

Abstract

This paper presents a novel pointer prefetching technique, called *multi-chain prefetching*. Multi-chain prefetching tolerates serialized memory latency commonly found in pointer chasing codes via aggressive prefetch scheduling. Unlike conventional prefetching techniques that hide memory latency underneath a single traversal loop or recursive function exclusively, multi-chain prefetching initiates prefetches for a chain of pointers prior to the traversal code, thus exploiting "pre-loop" work to help overlap serialized memory latency. As prefetch chains are scheduled increasingly early to accommodate long pointer chains, multi-chain prefetching overlaps prefetches across multiple independent linked structures, thus exploiting the natural memory parallelism that exists between separate pointer-chasing loops or recursive function calls.

This paper makes three contributions in the context of multi-chain prefetching. First, we introduce a prefetch scheduling technique that exploits pre-loop work and inter-chain memory parallelism to tolerate serialized memory latency. To our knowledge, our scheduling algorithm is the first of its kind to expose natural memory parallelism in pointer-chasing codes. Second, we present the design of a prefetch engine that generates a prefetch address stream at runtime, and issues prefetches according to the prefetch schedule computed by our scheduling algorithm. Finally, we conduct an experimental evaluation of multi-chain prefetching using six pointer-chasing applications, and compare it against an existing technique, jump pointer prefetching.

Our results show that multi-chain prefetching is an effective latency tolerance technique for pointer-chasing applications. On the four most memory-bound applications in our suite, multi-chain prefetching reduces execution time between 40% and 66%, and by 2.1% and 3.6% for the other two applications, compared to no prefetching. Multi-chain prefetching also outperforms jump pointer prefetching across all six of our applications, reducing execution time between 24% and 64% for the memory-bound applications, and by 2.1% and 4.9% for the other two applications, compared to jump pointer prefetching. Finally, multi-chain prefetching achieves its performance advantages without using jump pointers; hence, it does not require the intrusive code transformations necessary to create and manage jump pointer state.

1 Introduction

Prefetching, whether using software [12, 13, 9, 2], hardware [4, 14, 6, 7], or hybrid [18, 3, 5] techniques, has proven successful at hiding memory latency for applications that employ regular data structures (*e.g.* arrays). Unfortunately, these techniques are far less successful for pointer-intensive applications that employ linked data structures (LDSs) due to the memory serialization effects associated with LDS traversal, known as the *pointer chasing problem.* The memory operations performed for array traversal can issue in parallel because individual array elements can be referenced independently. In contrast, the memory operations performed for LDS traversal must dereference a series of pointers, a purely sequential operation. The lack of *memory parallelism* prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain, thus limiting their effectiveness for pointer-intensive applications.

Recently, researchers have begun investigating novel prefetching techniques for LDS traversal [8, 17, 16, 11, 10]. These new techniques address the pointer-chasing problem using one of two different approaches. Techniques in the first approach [16, 11, 10], which we call *stateless techniques*, perform prefetching using only the natural pointers belonging to the LDS. Existing stateless techniques prefetch pointer chains sequentially, so they do not exploit any memory parallelism (or they exploit only limited forms of memory parallelism [16]). Instead, they schedule each prefetch as early in the loop iteration as possible to maximize the amount of work available for memory latency overlap. Current stateless techniques can effectively tolerate memory latency only when the loops (or recursive functions) used to traverse the LDS contain sufficient work to hide the serialized memory latency. Unfortunately, many important pointer-chasing applications consist mostly of traversal loops with small amounts of work [8].

Techniques in the second approach [8, 17, 10], which we call *jump pointer techniques*, insert additional pointers into the LDS to connect non-consecutive link elements. These "jump pointers" allow prefetch instructions to name link elements further down the pointer chain without sequentially traversing the intermediate links, thus creating memory parallelism along a single chain of pointers. Jump pointer techniques achieve higher performance than existing stateless techniques [10]. Because they create memory parallelism using jump pointers, they can tolerate pointer-chasing cache misses even when the traversal loops contain insufficient work to hide the serialized memory latency. However, jump pointer techniques must create and manage the prefetch state, which incurs runtime and memory overhead and requires intrusive code transformations.

In this paper, we propose a new stateless prefetching technique called *multi-chain prefetching.* Like existing stateless techniques, multi-chain prefetching prefetches a single chain of pointers sequentially; however, it schedules the initiation of the chain of prefetches *prior* to the code that traverses the chain, thus overlapping a portion of the serialized memory latency with pre-loop work. As prefetch chains are scheduled increasingly early to accomodate long pointer chains, multi-chain prefetching overlaps prefetches across multiple independent linked structures, thus exploiting the natural memory parallelism that exists between separate pointer-chasing loops or recursive function calls. Because multi-chain prefetching hides cache miss latency underneath work outside of a single traversal, it tolerates serialized memory latency even when the traversal code contains very little work. Consequently, multi-chain prefetching achieves higher performance than previous stateless techniques. Furthermore, multi-chain prefetching does not use jump pointers. As a result, it does not suffer the overhead and code transformation penalties associated with jump pointer techniques.

Our work makes three contributions in the context of multi-chain prefetching. First, we introduce a prefetch scheduling technique that exploits pre-loop work and inter-chain memory parallelism to tolerate serialized memory latency. Our scheduling technique demonstrates that natural memory parallelism exists in many unmodified pointer-chasing applications. To our knowledge, our work is the first to expose such memory parallelism and to exploit it for the purpose of memory latency tolerance. Second, we present the design of a prefetch engine that generates a prefetch address stream at runtime, and issues prefetches according to the prefetch schedule computed by our scheduling algorithm. Finally, we conduct an experimental evaluation of multi-chain prefetching using six pointer-chasing applications, and compare it against jump pointer prefetching [10].

Our results show that multi-chain prefetching reduces execution time between 40% and 66% for four memory-bound applications, and by 2.1% and 3.6% for two other applications, compared to no prefetching. We also show that multi-chain prefetching outperforms jump pointer prefetching across all six of our applications, reducing execution time between 24% and 64% for the memory-bound applications, and by 2.1% and 4.9% for the other two applications, compared to jump pointer prefetching. Finally, we qualitatively argue that multi-chain prefetching is less intrusive than jump pointer prefetching, resulting in lower programming effort.

The rest of this paper is organized as follows. Section 2 further explains the essence of our approach. Section 3 presents our scheduling technique. Section 4 introduces our prefetch engine, and Section 5 presents experimental results. Finally, Section 6 concludes the paper.

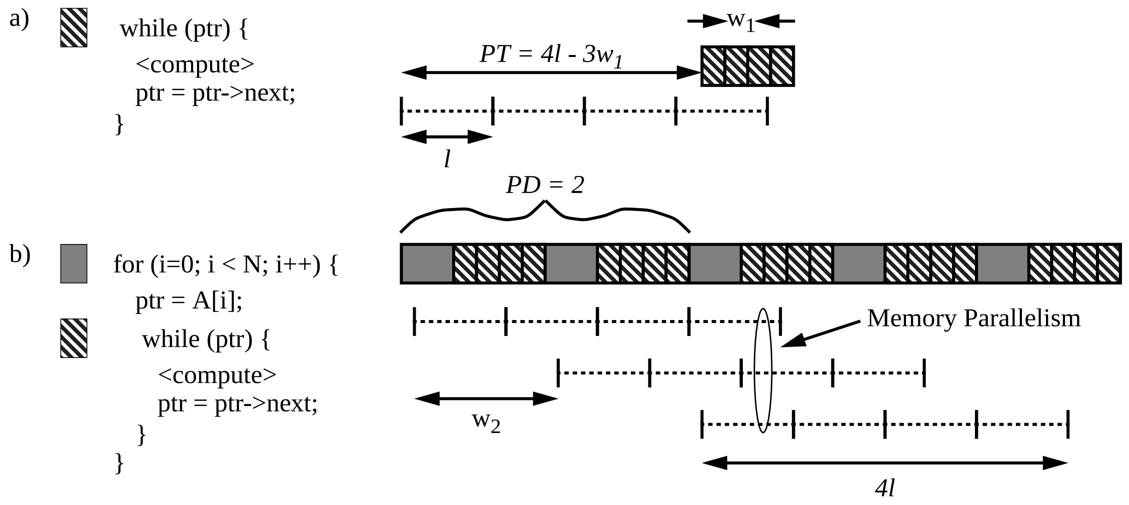


Figure 1: Overlapping pointer-chasing chains. a). Initiate prefetching of pointer chain prior to traversal loop. b). Exploit memory parallelism between independent pointer-chasing traversals.

2 Multi-Chain Prefetching

To illustrate the key idea behind multi-chain prefetching, consider a linked list traversal as shown in Figure 1a. In our example, a list of length four is traversed in a loop, denoted by a hashed box containing w_1 cycles of work. We assume an l-cycle cache miss latency, where $l > w_1$.

Figure 1a illustrates how multi-chain prefetching schedules the prefetch requests for the list nodes assuming all four prefetches miss in the cache. Because multi-chain prefetching uses the natural pointers in the LDS for prefetching, the prefetch requests must perform serially across $4l$ cycles, as represented by the dotted bars in Figure 1a. Since $l > w_1$, the loop alone contains insufficient work to hide the serialized memory latency. However, multi-chain prefetching tolerates all the memory latency by initiating the prefetch chain $4l - 3w_1$ cycles prior to the traversal loop, called the *pre-loop time* (PT). As illustrated in Figure 1a, these prefetches issue asynchronously with respect to the loop code. Our technique relies on a prefetch engine to issue asynchronous prefetches independent of the main CPU.

While pre-loop overlap tolerates the serialized memory latency for a single pointer chain, pointer chasing computations typically traverse many pointer chains, each of which is often independent. To illustrate how multi-chain prefetching exploits such independent pointer-chasing traversals, Figure 1b shows a doubly nested loop that traverses an array of lists. The outer loop, denoted by a shaded box with w_2 cycles of work, traverses an array that extracts a root pointer for the inner loop. The inner loop is identical to the loop in Figure 1a.

As in Figure 1a, multi-chain prefetching still initiates each chain of prefetches PT cycles prior to its corresponding inner loop. To provide the PT cycles of pre-inner loop overlap, each prefetch chain is initiated from an earlier outer loop iteration. The number of outer loop iterations by which a prefetch chain must be initiated in advance is known as the *prefetch distance* (PD). Notice when $PD > 0$, the prefetches from separate chains overlap across iterations of the outer loop, exposing memory parallelism despite the fact that each chain is prefetched serially. As illustrated in Figure 1b, the initiation of prefetch chains occurs synchronously with iterations of the outer loop. Our prefetch engine synchronizes with the processor to initiate such synchronous prefetch chains.

Figure 1 illustrates that the success of multi-chain prefetching relies on early knowledge of the pointer chain addresses. An important question is whether these addresses can be determined sufficiently early. For applications with static traversal patterns, static analysis can provide the pointer chain addresses at any time. For applications with data-dependent traversal patterns, the pointer chain addresses may not be known until soon before the traversal code is reached. In these applications, code transformations may be required to compute pointer chain addresses early, if possible; otherwise, early prefetch initiation will be limited. The primary goal of this paper is to explore the potential for exploiting natural memory parallelism in pointer-chasing applications; therefore, our preliminary investigation of multi-chain prefetching focuses on static traversal patterns. Further research is required to handle data-dependent traversal patterns.

3 Prefetch Chain Scheduling

In this section, we describe how to schedule prefetch chains in multi-chain prefetching. We present our technique in two parts. First, Section 3.1 introduces an LDS descriptor framework that captures the information required for prefetch chain scheduling. Second, Section 3.2 describes a scheduling algorithm that computes the scheduling parameters (PT and PD) from the LDS descriptors.

3.1 LDS Descriptor Framework

To perform prefetch chain scheduling, we first analyze the LDS traversal code and extract two types of information: data structure layout, and traversal code work. The data structure layout information captures the dependences between memory references, thus identifying pointer-chasing chains. This information is also used by the prefetch engine to generate prefetch addresses at

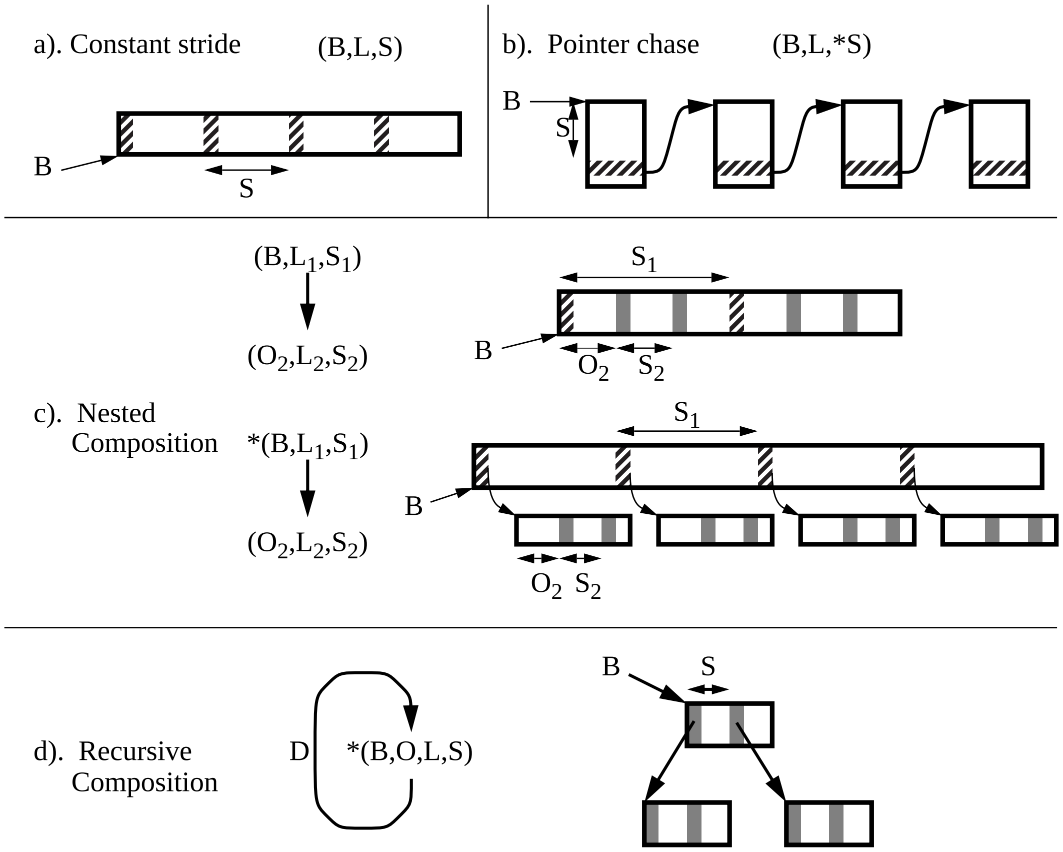


Figure 2: LDS descriptors: data layout information.

runtime (see Section 4). Along with the data structure layout information, the traversal code work information enables our scheduling algorithm, presented later in Section 3.2, to compute the scheduling parameters. In the rest of this section, we define an *LDS descriptor framework* used to specify both types of information.

Data structure layout is specified using two descriptors, one for arrays and one for linked lists. Figure 2a illustrates the array descriptor which contains three parameters: base (B), length (L), and stride (S). These parameters specify the base address of the array, the number of array elements traversed by the application code, and the stride between consecutive memory references, respectively, and represent the memory reference stream for a constant-stride array traversal. Figure 2b illustrates the linked list descriptor which contains three parameters similar to the array descriptor. For the linked list descriptor, the B parameter specifies the root pointer of the list, the L parameter specifies the number of link elements traversed by the application code, and the $*S$ parameter specifies the offset from each link element address where the "next" pointer is located.

To specify the layout of complex data structures, our framework permits descriptor composition.

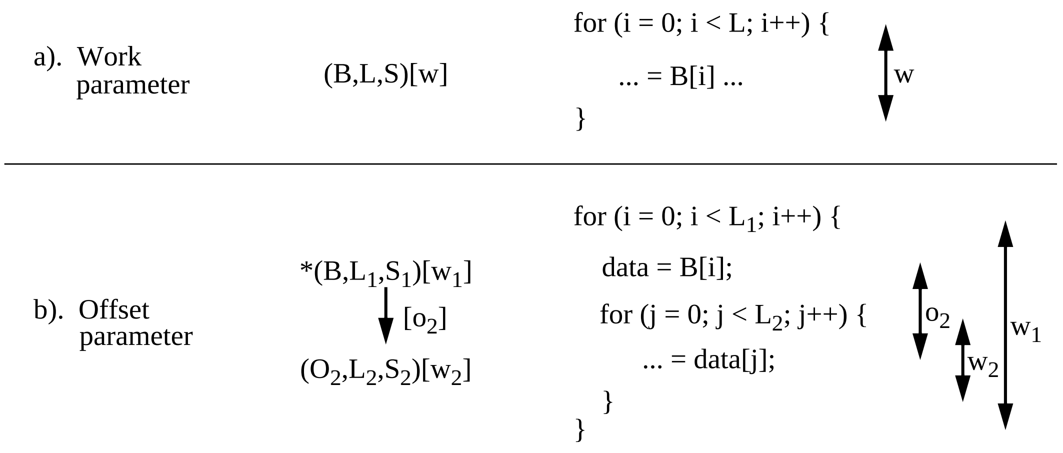


Figure 3: LDS descriptors: traversal code work information.

Composition is represented as a directed graph whose nodes are array or linked list descriptors, and whose edges denote address generation dependences. Two types of composition are allowed: nested and recursive. In a nested composition, each address generated by an outer descriptor forms the B parameter for multiple instantiations of a dependent inner descriptor. A parameter, O, can be specified to shift the base address of each inner descriptor by a constant offset. Such nested descriptors capture the data access patterns of nested loops. In the top half of Figure 2c, we show a nested descriptor corresponding to the traversal of a 2-D array. We also permit indirection between nested descriptors denoted by a "*" in front of the outer descriptor, as illustrated in the lower half of Figure 2c. Although the examples in Figure 2c only use a single descriptor at each nesting level, our framework permits multiple inner descriptors to be nested underneath a single outer descriptor.

In addition to nested composition, our framework also permits recursive composition. Recursively composed descriptors are identical to nested descriptors, except the dependence edge flows backwards. Since recursive composition introduces cycles into the descriptor graph, our framework requires each recursive arc to be annotated with the depth of recursion, D, to bound the size of the data structure. Figure 2d shows a simple recursive descriptor in which the inner and outer descriptors are one and the same, corresponding to a simple tree data structure. (Notice the D parameter, as well as the L parameter for linked lists, is used only for scheduling purposes. Our prefetch engine, discussed in Section 4, does not require information about the extent of dynamic data structures, and instead prefetches until it encounters a null-terminated pointer.)

Finally, Figure 3 shows the extensions to the LDS descriptors that provide the traversal code work information. The work information specifies the amount of work performed by the applica-

```
for (i =  N-1 down to 0) {
    $PTnest_i = \max_{\text{composed k via indirection}} (PT_k - o_k)$          (1)
    if ((descripor i is pointer-chasing) and ($l > w_i$)) {
        $PT_i = L_i * (l - w_i) + w_i + PTnest_i$                  (2)
        $PD_i = \infty$;                                           (3)
    } else {
        $PT_i = l + PTnest_i$                                      (4)
        $PD_i = \lceil PT_i / w_i \rceil$                          (5)
    }
}
```

Figure 4: Scheduling algorithm for static descriptor graphs.

tion code as it traverses the LDS. To provide the work information, the LDS descriptor graph is annotated with two types of parameters. The *work* parameter, w, specifies the amount of work per traversal loop iteration. Shown in Figure 3a, the work parameter annotates each array or linked list descriptor. The *offset* parameter, o, is used for composed descriptors, and specifies the amount of work separating the first iteration of the inner descriptor from each iteration of the outer descriptor. Shown in Figure 3b, the offset parameter annotates each composition dependence edge.

3.2 Scheduling Algorithm

Once an LDS descriptor graph has been constructed from a traversal code, we compute a prefetch chain schedule from the descriptor graph. This section presents the algorithm that computes the scheduling information. Section 3.2.1 describes our basic scheduling algorithm assuming the descriptor graph contains descriptor parameters that are all statically known. Then, Section 3.2.2 briefly describes how our scheduling algorithm handles graphs with unknown descriptor parameters.

3.2.1 Static Descriptor Graphs

Our scheduling algorithm computes three scheduling parameters for each descriptor i in the descriptor graph: whether the descriptor requires asynchronous or synchronous prefetching, the pre-loop time, PT_i, and the prefetch distance, PD_i (see Section 2 for an overview of these parameters). Figure 4 presents our scheduling algorithm. The algorithm is defined recursively, and processes descriptors from the leaves of the descriptor graph to the root. The "for ($N-1$ down to 0)" loop processes the descriptors in the required bottom-up order assuming we assign a number between 0 and $N-1$ in top-down order to each of the N descriptors in the graph. Our scheduling algorithm assumes there are no cycles in the graph. A cyclic graph, arising from recursively composed descriptors, must first be converted into an acyclic graph before applying our scheduling algorithm (see Figure 5). Our scheduling algorithm also assumes the cache miss latency to physical memory, l, is known.

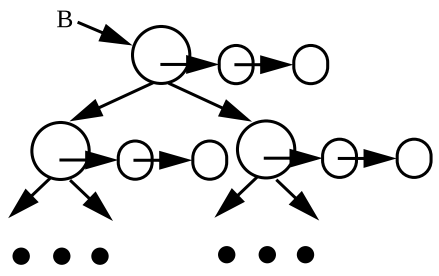
We now describe the computation of the three scheduling parameters for each descriptor visited in the descriptor graph. Descriptor i requires asynchronous prefetching if it traverses a linked list and there is insufficient work in the traversal loop to hide the serialized memory latency (*i.e.* $l > w_i$). Otherwise, if descriptor i traverses an array or if $l \leq w_i$, then it requires synchronous prefetching.[1] The "if" conditional test in Figure 4 computes whether asynchronous or synchronous prefetching is used.

Next, we compute the pre-loop time, PT_i. For asynchronous prefetching, we must overlap that portion of the serialized memory latency that cannot be hidden underneath the traversal loop itself with work prior to the loop. Figure 1 shows $PT_1 = 4l - 3w_1$ for a 4-iteration pointer-chasing loop. In general, $PT_i = L_i * (l - w_i) + w_i$. For synchronous prefetching, we need to only hide the cache miss for the first iteration of the traversal loop, so $PT_i = l$. Equations 2 and 4 in Figure 4 compute PT_i for asynchronous and synchronous prefetching, respectively. Notice these equations both contain an extra term, $PTnest_i$. $PTnest_i$ serializes PT_i and PT_k, the pre-loop time for any nested descriptor k composed via indirection (see lower half of Figure 2c). Serialization occurs between composed descriptors that use indirection because of the data dependence caused by indirection. We must sum PT_k into PT_i; otherwise, the prefetches for descriptor k will not initiate early enough. Equation 1 in Figure 4 considers all descriptors composed under descriptor i that use indirection and sets $PTnest_i$ to the largest PT_k found. The offset, o_k, is subtracted because it overlaps with descriptor k's pre-loop time.

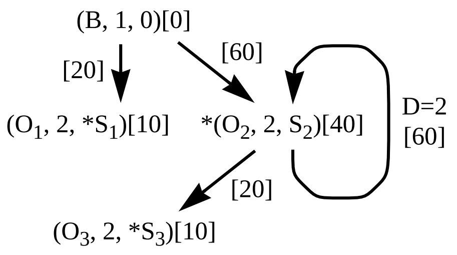
Finally, we compute the prefetch distance, PD_i. Descriptors that require asynchronous prefetching do not have a prefetch distance; we denote this by setting $PD_i = \infty$. The prefetch distance for descriptors that require synchronous prefetching is exactly the number of loop iterations necessary to overlap the pre-loop time, which is $\lceil \frac{PT_i}{w_i} \rceil$. Equations 3 and 5 in Figure 4 compute the prefetch distance for asynchronous and synchronous prefetching, respectively.

Figure 5 presents a detailed scheduling example using an LDS traversal that resembles the traversal in Health, one of the applications from our experimental evaluation. The example illus-

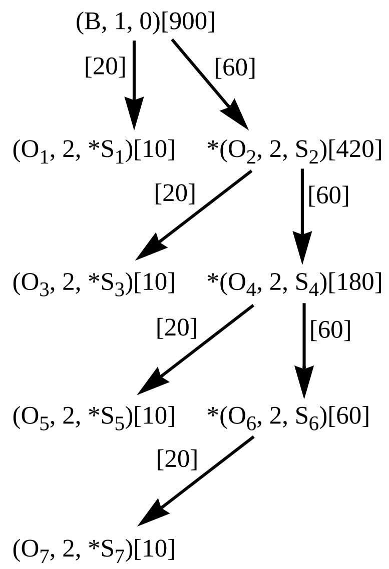
[1]This implies that linked list traversals in which $l <= w_i$ use synchronous prefetching since prefetching one link element per loop iteration can tolerate the serialized memory latency when sufficient work exists in the loop code.



a). Tree-of-lists data structure. A balanced binary tree of depth 4 contains a linked list at each node of length 2. Assume a depth-first traversal of the tree in which each linked list is traversed before the recursive call to the left and right children.



b). Extract data layout and traversal code work information. Descriptor 0 is a "dummy" descriptor that provides the root address of the tree. Descriptors 1 and 2 are the linked list and tree node at the root of the tree. Composed under descriptor 2 is the linked list (descriptor 3 which is identical to descriptor 1) and tree node (descriptor 2 again) of the next lower level. The data layout information in this descriptor is exactly the information used in our prefetch engine, described in Section 4.



c). Convert cyclic graph into acyclic graph. "Unroll" all recursively composed descriptors by copying the descriptors involved in each cycle by the depth of the recursion, D. In our example, descriptors 4 and 6 are copies of descriptor 2, and descriptors 5 and 7 are copies of descriptor 3. After copying, update the work and offset parameters to reflect the additional work created by the copies.

$PT_7 = 2*(76-10)+10 = \mathbf{142}$

$PD_7 = \infty$

$PT_6 = 76+PT_7-20 = \mathbf{198}$

$PD_6 = \lceil PT_6 / 60 \rceil = \mathbf{4}$

$PT_5 = 2*(76-10)+10 = \mathbf{142}$

$PD_5 = \infty$

$PT_4 = 76+PT_6-60 = \mathbf{214}$

$PD_4 = \lceil PT_4 / 180 \rceil = \mathbf{2}$

$PT_3 = 2*(76-10)+10 = \mathbf{142}$

$PD_3 = \infty$

$PT_2 = 76+PT_4-60 = \mathbf{230}$

$PD_2 = \lceil PT_2 / 420 \rceil = \mathbf{1}$

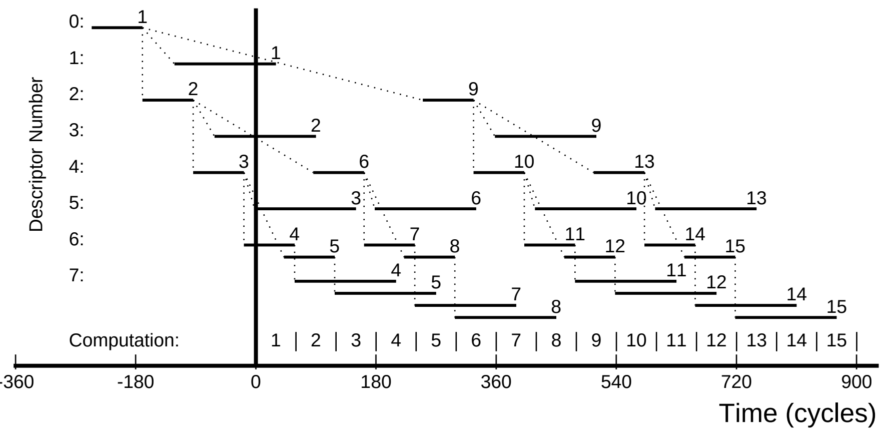
$PT_1 = 2*(76-10)+10 = \mathbf{142}$

$PD_1 = \infty$

$PT_0 = 76+PT_2-60 = \mathbf{246}$

$PD_0 = \lceil PT_0 / 900 \rceil = \mathbf{1}$

d). Compute scheduling parameters. Using our scheduling algorithm in Figure 4, compute the pre-loop time, *PT_i*, and the prefetch distance, *PD_i*, for each descriptor in bottom-up order. This solution assumes *l=76*.



e). Prefetch schedule for the tree-of-lists example. Short solid horizontal lines denote prefetches for the tree nodes, long solid horizontal lines denote 2 serialized prefetches for the linked lists, and dotted lines indicate address generation dependences. The "Computation" timeline indicates the traversal of the tree nodes by the application code. Each prefetch has been labeled with the number of the traversal that consumes the prefetched data. The figure shows our scheduling algorithm exposes significant memory parallelism.

Figure 5: Scheduling example. a). The tree-of-lists data structure used in our example (similar to the data structure in Health). b). Extraction of data layout and traversal code work information. c). Conversion of the cyclic graph into an acyclic graph. d). Computation of the scheduling parameters. e). Graphical representation of the prefetch schedule.

trates how the LDS descriptor graph is created, how cycles in the graph are eliminated, and how the scheduling algorithm computes the scheduling parameters.

3.2.2 Dynamic Descriptor Graphs

Section 3.2.1 describes our scheduling algorithm assuming the descriptor graph is static. In most descriptor graphs, the list length and recursion depth parameters are unknown. Because the compiler does not know the extent of dynamic data structures a priori, it cannot exactly schedule all the prefetch chains using our scheduling algorithm. However, we make the key observation that all prefetch distances in a dynamic graph are bounded, regardless of actual chain lengths. Consider the array-of-lists example from Figure 1. The prefetch distance of each linked list is $PD = \lceil PT/w_2 \rceil$. As the list length, L, increases, both PT and w_2 increase linearly. In practice, the ratio PT_i/w_i increases asymptotically to an upper bound value as pointer chains grow in length. Our scheduling algorithm can compute the bounded prefetch distance for all descriptors by substituting large values into the unknown parameters in the dynamic descriptor graph. Since bounded prefetch distances are conservative, they may initiate prefetches earlier than necessary. In Section 5, we will quantify this effect.

4 Prefetch Engine

Multi-chain prefetching requires issuing prefetch requests prior to the traversal loop (see Figure 1). Due to the asynchronous nature in which such pre-loop pointer-chasing prefetches are scheduled, it is difficult or impossible to software pipeline them into the traversal code. In this section, we introduce a programmable prefetch engine that dynamically issues the prefetches outside of the main CPU given the data layout information described in Section 3.1 and the scheduling parameters computed by our scheduling algorithm.

The hardware organization of the prefetch engine appears in Figure 6. The design requires three additions to a commodity microprocessor: a prefetch buffer [9], the prefetch engine itself, and two new instructions called $SINIT$ and $SYNC$. The prefetch engine prefetches data into the prefetch buffer if it is not already in the L1 cache at the time the prefetch is issued (a prefetch from main memory is placed in the L2 cache on its way to the prefetch buffer). All processor memory accesses check both the cache and the prefetch buffer, and if a hit in the prefetch buffer occurs, the corresponding cache block is moved into the cache.

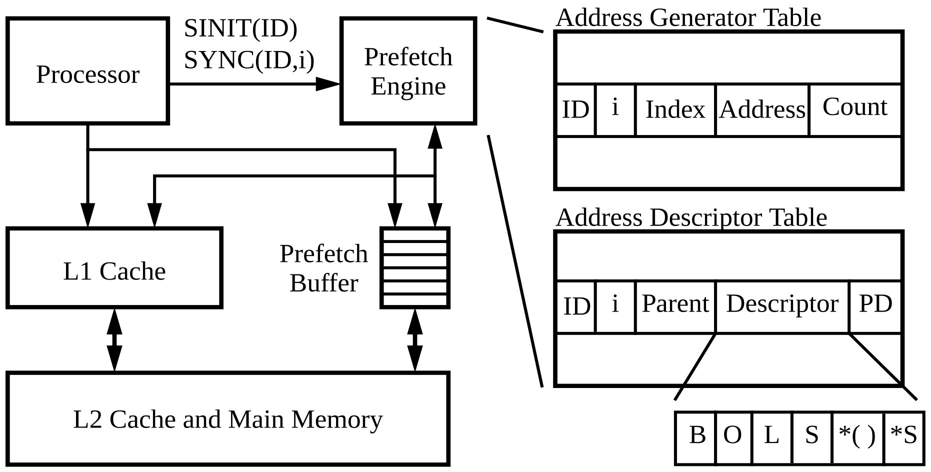


Figure 6: Prefetch engine hardware and integration with a commodity microprocessor.

In the next section, we describe how the prefetch engine hardware generates addresses from the LDS descriptors introduced in Section 3.1. Then, in Section 4.2, we discuss how the ISA extensions are used to help properly schedule prefetch requests.

4.1 Prefetch Address Generation

The prefetch engine consists of two hardware tables, the *Address Descriptor Table* (ADT), and the *Address Generator Table* (AGT). The ADT stores the data layout information from the LDS descriptors described in Section 3.1. Each array or linked list descriptor in an LDS descriptor graph occupies a single ADT entry, identified by the graph number, ID (each LDS descriptor graph is assigned a unique ID), and the descriptor number, i, assuming the top-down numbering of descriptors discussed in Section 3.2.1. The *Parent* field specifies descriptor i's parent in the descriptor graph. The next four values, B, O, L, and S, specify the descriptor parameters illustrated in Figure 2.[2] $*()$ is a 1-bit value that specifies whether indirection is used between descriptor i and its parent, and $*S$ is another 1-bit value that specifies whether the descriptor is an array or linked list descriptor. Finally, the PD field stores the prefetch distance computed by our scheduling algorithm for descriptor i. The ADT is memory mapped into the main processor's address space, and is written during program initialization.

The AGT generates the prefetch address stream specified by the data layout information stored in the ADT. AGT entries are activated dynamically as prefetch addresses are generated. Once

[2]The prefetch engine assumes all lists have infinite length and recursions have infinite depth. At runtime, the prefetch engine prefetches until it encounters a null-terminated pointer to indicate the end of a list or recursive structure.

activated, an AGT entry generates the prefetch addresses for a single LDS descriptor. AGT entry activation can occur in two ways. First, the main processor can execute an $SINIT(ID)$ instruction to initiate prefetching for the data structure identified by ID. The prefetch engine searches the ADT for the entry matching ID and $i = 0$ (*i.e.* the root node for descriptor graph ID). An AGT entry is allocated for this descriptor, the $Index$ field is set to zero, and the $Address$ field is set to the B parameter in ADT entry $i = 0$. Second, when an active AGT entry, i, computes a new prefetch address (explained below), a new AGT entry, j, is activated for every node in the descriptor graph that is a child of node i. This is done by searching the ADT for all entries, j, with $Parent = i$. For each new AGT entry j, $Index$ is set to zero, and the new address generated by AGT entry i is placed in the $Address$ field of AGT entry j. If the $*()$ bit in ADT entry j is set, indicating indirection between descriptors i and j, then the prefetch engine issues a load for the $Address$ field in AGT entry j to perform the indirection, and places the data value loaded back in j's $Address$ field. Finally, the O value from ADT entry j is added to the $Address$ field to form the final base address.

After an AGT entry i is activated and its base address computed, the memory addresses associated with descriptor i are generated one at a time. Each new address is computed by adding the S parameter in ADT entry i to the current value in AGT entry i's $Address$ field. In addition, the $Index$ field is incremented. If the $*S$ bit in ADT entry i is set, indicating pointer chasing, then the prefetch engine issues a load for the $Address$ field in AGT entry i to perform the indirection, and places the data value loaded back in i's $Address$ field. Whenever an AGT entry generates a new memory address, it must wait until the prefetch for the address is issued before generating the next address in the sequence (explained below). Finally, when the $Index$ field in AGT entry i reaches the L parameter in ADT entry i, the AGT entry is deactivated.

4.2 Prefetch Scheduling

When an active AGT entry i generates a new memory address, the prefetch engine must schedule a prefetch for the memory address. Prefetch scheduling occurs in one of two ways. First, if the prefetches for descriptor i should issue asynchronously (*i.e.* $PD_i = \infty$), the prefetch engine issues a prefetch for AGT entry i immediately after a new memory address is computed in the AGT entry. Consequently, prefetches for asynchronous AGT entries traverse a pointer chain as fast as possible, throttled only by the serialized cache misses that occur along the chain.

Second, if the prefetches for descriptor i should issue synchronously (*i.e.* $PD_i \neq \infty$), then the prefetch engine synchronizes the prefetches for AGT entry i with the code that traverses the corresponding array or linked list. We rely on the compiler or programmer to insert a $SYNC$ instruction at the top of the loop or recursive function call that traverses the data structure to provide the synchronization information. Furthermore, the prefetch engine must maintain the proper prefetch distance, as computed by our scheduling algorithm, for such synchronized AGT entries. A $Count$ field in the AGT entry is used to maintain this prefetch distance. The $Count$ field is initialized to the PD value in the ADT entry (computed by the scheduling algorithm) upon initial activation of the AGT entry. The $Count$ value is decremented each time the prefetch engine issues a prefetch for the AGT entry. In addition, the prefetch engine "listens" for $SYNC$ instructions. When a $SYNC$ executes, it emits both an ID and an i value that matches an AGT entry. On a match, the $Count$ value in the matched AGT entry is incremented. The prefetch engine issues a prefetch as long as $Count > 0$. Once $Count$ reaches 0, the prefetch engine waits for the $Count$ value to be incremented before issuing the prefetch for the AGT entry, which occurs the next time the corresponding $SYNC$ instruction executes.

5 Results

In this section, we conduct an evaluation of multi-chain prefetching and compare it to jump pointer prefetching. After describing our experimental methodology in Section 5.1, Section 5.2 presents the main results for multi-chain prefetching. Then, Section 5.3 examines the prefetching results more closely to further improve performance for both multi-chain prefetching and jump pointer prefetching. Finally, Section 5.4 examines the programming effort required by our technique.

5.1 Experimental Methodology

To evaluate the performance of multi-chain prefetching, we constructed a detailed event-driven simulator of the prefetch engine architecture described in Section 4 coupled with a state-of-the-art RISC processor. Our simulator uses the processor model from the SimpleScalar Tool Set [1] to model a dynamically scheduled 4-way issue processor. For the L1 cache, we use a split 16-Kbyte instruction/16-Kbyte data direct-mapped write-through cache with a 32-byte block size. For the L2 cache, we use a unified 512-Kbyte 4-way set-associative write-back cache with a 64-byte block size. The L1 and L2 caches have 8 and 16 MSHRs, respectively, to enable significant

Application	Data Structure	Input Parameters
EM3D	array of pointers	10,000 nodes
MST	list of lists	1024 nodes
Treeadd	balanced binary tree	20 levels
Health	balanced quadtree of lists	5 levels, 500 iters
Perimeter	unbalanced quadtree	4K x 4K image
Bisort	balanced binary tree	250,000 numbers

Table 1: Benchmark summary.

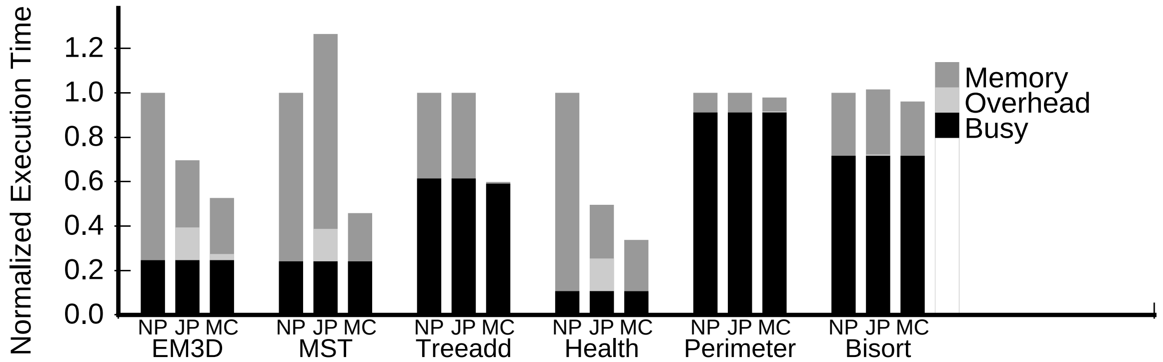


Figure 7: Execution time for no prefetching (NP), jump pointer prefetching (JP), and multi-chain prefetching (MC). Each execution time bar has been broken down into useful cycles (Busy), prefetch-related cycles (Overhead), and memory stall cycles (Memory).

memory concurrency. We also assume an aggressive memory sub-system with 8.5 Gbytes/sec peak bandwidth and 64 banks. Bus and bank contention are faithfully simulated. Finally, we assume the prefetch engine clocks at the same rate as the processor, and has a 1-Kbyte prefetch buffer that is accessed in parallel with the L1 cache. Access to the L1 cache/prefetch buffer, L2 cache, and main memory costs 1 cycle, 10 cycles, and 76 cycles respectively. To drive our simulations, we use six applications from the Olden benchmark suite [15]. Table 1 summarizes the applications.

5.2 Multi-Chain Prefetching Performance

Figure 7 presents the results of multi-chain prefetching for our six applications. For each application, we report the execution time without prefetching, labeled "NP," with jump pointer prefetching, labeled "JP," and with multi-chain prefetching, labeled "MC." For the JP bars, we follow the algorithm proposed in [10]. Jump pointers are inserted into all prefetched link nodes and are updated on the first traversal of the LDS using a FIFO queue of history pointers (see [10] for more details). Each bar in Figure 7 has been broken down into three components: time spent executing useful instructions, time spent executing prefetch-related instructions, and time spent in memory stall, labeled "Busy," "Overhead," and "Memory," respectively.[3] All times have been normalized against the NP bar for each application.

Comparing the MC bars versus the NP bars, multi-chain prefetching eliminates a significant fraction of the memory stall, reducing overall execution time from 40% to 66% for the four most memory-bound applications (EM3D, MST, Treeadd, and Health), and 2.1% and 3.6% for the other two applications (Perimeter and Bisort). Due to memory serialization and the small amounts of work in the LDS traversal codes, out-of-order execution cannot tolerate the cache misses suffered in these applications. However, multi-chain prefetching successfully overlaps most of the memory latency by exploiting pre-loop overlap and memory parallelism across independent pointer chains.

Comparing the MC bars versus the JP bars, multi-chain prefetching outperforms jump pointer prefetching on all six applications, reducing execution time between 24% to 64% for the four memory-bound applications, and 2.1% and 4.9% for the other two applications. Two major factors contribute to multi-chain prefetching's performance advantage: reduced software overhead, and increased cache miss coverage. In the rest of this section, we examine these two major factors in greater detail to better understand the advantages of multi-chain prefetching.

Multi-chain prefetching incurs lower software overhead as compared to jump pointer prefetching for EM3D, MST, and Health. In MST, jump pointer prefetching suffers high jump pointer creation overhead. On the first traversal of an LDS, jump pointer prefetching must create jump pointers for prefetching subsequent traversals; consequently, applications that perform a small number of LDS traversals spend a large fraction of time in jump pointer creation code. In MST, the linked list structures containing jump pointers are traversed only 4 times, resulting in overhead that costs 60% as much as the traversal code itself. In addition to jump pointer creation overhead, jump pointer prefetching also suffers jump pointer management overhead. Applications that modify the LDS during execution require fix-up code to keep the jump pointers consistent as the LDS changes. Health performs frequent link node insert and delete operations. In Health, jump pointer fix-up code is responsible for most of the 137% increase in the traversal code cost.[4] Since multi-chain prefetching only uses natural pointers for prefetching, it does not suffer any jump pointer creation or management overheads.

The jump pointer prefetching version of EM3D suffers high prefetch instruction overhead. Jump

[3]The "Busy" components were obtained by running separate simulations assuming a perfect memory system.

[4]Like Health, Bisort also modifies its LDS during execution. However, the modifications are complex, and it is unclear how to update the jump pointers. Consequently, we do not attempt to update the jump pointers in Bisort.

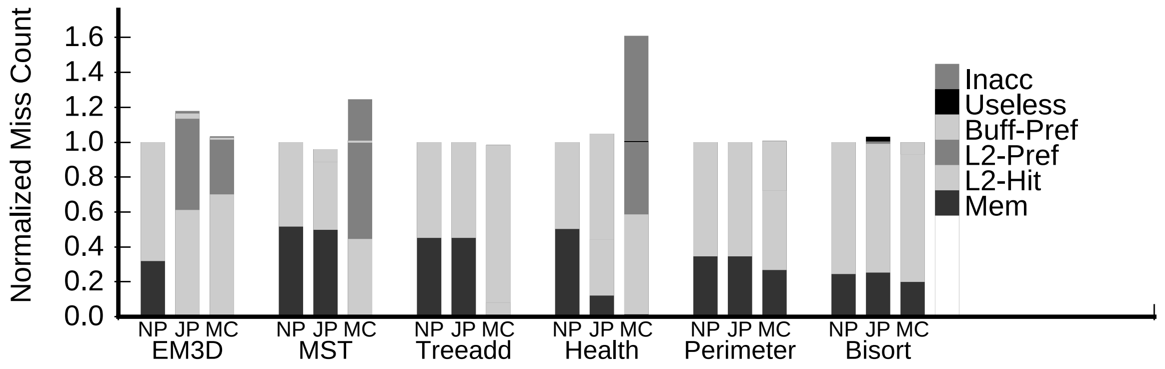


Figure 8: Cache miss breakdown. Each bar has been broken down into misses to main memory (Miss), hits in the L2 (L2-Hit), and additional L2 (L2-Pref) and prefetch buffer (Buff-Pref) cache hits due to prefetching. Prefetches that did not contribute to increased hits are useless (Useless), or inaccurate (Inacc).

pointer prefetching inserts address computation and prefetch instructions into the application code. In multi-chain prefetching, this overhead is off-loaded onto the prefetch engine (at the expense of hardware support). While multi-chain prefetching requires the use of $SYNC$ and $SINIT$ instructions, these instructions introduce negligible overhead. In EM3D, the traversal loop is inexpensive, hence the added code in jump pointer prefetching dilates the loop cost by 60%. Prefetch instructions also contribute to the software overheads visible in MST and Health.

In addition to reduced software overhead, multi-chain prefetching also achieves higher cache miss coverage than jump pointer prefetching, which accounts for the improved memory stall reduction in MST, Treeadd, Perimeter, and Bisort in Figure 7. To illustrate this point, Figure 8 shows a breakdown of cache misses for our simulations. The NP bars in Figure 8 break down the L1 cache misses without prefetching into misses satisfied from the L2 cache, labeled “L2-Hit,” and misses satisfied from main memory, labeled “Mem.” The JP and MC bars show the same two components, but in addition, also show the increase in L2 cache hits and the addition of prefetch buffer hits due to prefetching, labeled “L2-Pref” and “Buff-Pref,” respectively. Figure 8 also shows prefetch requests that did not improve cache miss coverage. Useless prefetches, labeled “Useless,” are evicted from both the prefetch buffer and the L2 cache before being accessed by the processor because they arrive too early, and inaccurate prefetches, labeled “Inacc,” are prefetches that are never accessed. All bars are normalized against the NP bar for each application.

Multi-chain prefetching achieves higher cache miss coverage for Treeadd, Perimeter, and Bisort due to first-traversal prefetching. In multi-chain prefetching, all LDS traversals can be prefetched. Jump pointer prefetching, however, is ineffective on the first traversal because it must create the jump pointers before it can perform prefetching. For Treeadd and Perimeter, the LDS is traversed only once, so jump pointer prefetching does not perform any prefetching. In Bisort, the LDS is traversed twice, so prefetching is performed on only half the traversals. In contrast, multi-chain prefetching converts 90%, 28%, and 8% of the original cache misses into prefetch buffer hits for Treeadd, Perimeter, and Bisort, respectively. Figure 7 shows an execution time reduction of 40%, 2.1%, and 3.6% for these applications.[5]

In MST, multi-chain prefetching achieves higher cache miss coverage due to early link node prefetching. Multi-chain prefetching prefetches all link nodes in a pointer chain, while jump pointer prefetching does not prefetch the first PD (prefetch distance) nodes because there are no jump pointers that point to these early nodes. In MST, jump pointer prefetching leaves most link nodes unprefetched because the linked lists are very short. Only 14.3% of MST's original cache misses are converted into prefetch buffer hits by jump pointer prefetching. In contrast, multi-chain prefetching converts all of MST's "Mem" component into L2 hits. Figure 7 shows an execution time reduction of 54%. However, notice that multi-chain prefetching is unable to convert any cache misses into prefetch buffer hits for MST. We will explain this effect in Section 5.3.

In Health, jump pointer prefetching converts 62% of the original cache misses into prefetch buffer hits, but 38% of the original cache misses remain. Most important, 32% of the misses after prefetching are misses all the way to main memory. Again, this is due to jump pointer prefetching's inability to prefetch early link nodes. Multi-chain prefetching prefetches all the link nodes, resulting in zero misses to main memory; however, like MST, multi-chain prefetching does not convert any cache misses into prefetch buffer hits. The net effect is that the JP and MC bars in Figure 7 have roughly equal memory stall components.

Recently, Karlsson et al [8] proposed an extension to jump pointer prefetching, called *prefetch arrays*, that enables prefetching for early nodes by adding pointers at the beginning of a pointer chain. A comparison against prefetch arrays is beyond the scope of this paper. While prefetch arrays will tend to equalize the cache miss coverage between JP and MC, we note that prefetch arrays still cannot hide the cache miss to the first node in a pointer chain.

[5]First-traversal prefetching also benefits MST since this application traverses its linked lists only 4 times. However, early link node prefetching, explained in the next paragraph, limits the cache miss coverage achieved by jump pointer prefetching more than the lack of first-traversal prefetching for MST.

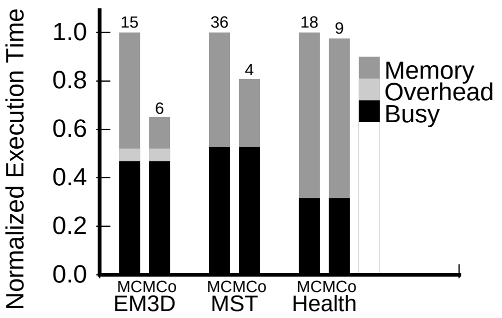


Figure 9: Performance with reduced prefetch distances.

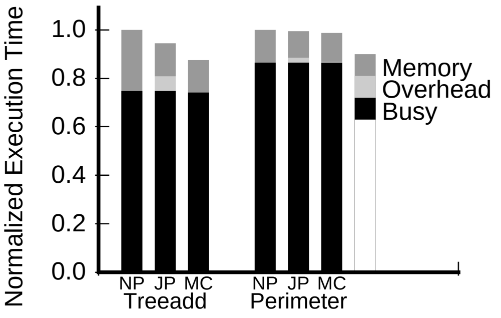


Figure 10: Creating jump pointers in LDS creation code for single-traversal applications.

5.3 Further Improving Performance

Figure 8 shows that multi-chain prefetching is unable to convert any of the original cache misses into prefetch buffer hits for EM3D, MST, and Health. This limitation in multi-chain prefetching is due to early prefetch arrival. Because multi-chain prefetching exploits pre-loop overlap, a large fraction of prefetches arrive before they are accessed and occupy the prefetch buffer. For applications with long pointer chains, the number of early prefetches can exceed the prefetch buffer capacity and cause thrashing. Since prefetched data is also placed in the L2 cache, the processor will normally enjoy an L2 hit where greater capacity prevents thrashing, but the L1-L2 latency is exposed.

While early prefetch arrival is a limitation in multi-chain prefetching, the number of early prefetches in our experiments is large due to overly conservative assumptions made by our scheduling algorithm. As discussed in Section 3.2, our scheduling algorithm computes a bounded prefetch distance to accommodate unknown list lengths and recursion depths, and it assumes all cache misses incur the full latency to physical memory. To see whether multi-chain prefetching can cover more misses in the prefetch buffer, we reduced the prefetch distance to an optimal value, determined experimentally for each application, to minimize the number of early prefetches. Figure 9 shows the result of this experiment. The bars labeled "MC" are the same as those from Figure 7, and the bars labeled "MCo" report execution time using the optimal prefetch distance. The prefetch distance for the most important LDS descriptor appears above each bar. Reducing the prefetch distance reduces multi-chain prefetching execution time by 35%, 19% and 2.5% for EM3D, MST, and Health, respectively. These results suggest that additional performance exists if our scheduling algorithm obtains more accurate information, either through better static analysis or through profiling.

As shown in Section 5.2, one limitation to jump pointer prefetching performance is the inability to prefetch on the first traversal, preventing any gains for single-traversal applications such as Treeadd and Perimeter. Figure 10 compares the performance of jump pointer prefetching and multi-chain prefetching for Treeadd and Perimeter when jump pointer creation occurs in the LDS creation code, thus enabling prefetching during LDS traversal. In these experiments, we report the execution times for both LDS creation and LDS traversal phases of the computation.

Figure 10 shows that jump pointer prefetching and multi-chain prefetching are equally effective at eliminating memory stall. The remaining memory stall after prefetching in both cases occur in the LDS creation code since neither technique prefetches during this phase of the application. However, due to the software overhead in jump pointer prefetching suffered for jump pointer creation, multi-chain prefetching still maintains a performance advantage of 7.4% and 0.74% for Treeadd and Perimeter, respectively.

5.4 Programming Effort

Jump pointer techniques are burdened by the creation and management of prefetch state, leading to several performance advantages for multi-chain prefetching as demonstrated in Section 5.2. In addition to performance advantages, the stateless nature of multi-chain prefetching also provides programming effort advantages. Compared to jump pointer techniques, multi-chain prefetching requires fewer code changes and less knowledge about the semantics of the program.

Jump pointer prefetching requires augmentation of the data structures to insert the jump pointers. It also requires insertion of jump pointer creation code to initialize jump pointer values on the first traversal of an LDS. For applications that traverse LDSs a small number of times, the jump pointer creation code may have to be integrated with the LDS creation code. This is more difficult, especially if the order of LDS creation differs from the order of LDS traversal. Finally, it must be determined if an LDS is modified after its creation, and if so, it must be determined where the modifications occur. Additional code must be inserted at the modification sites to perform the jump pointer updates.

In contrast, multi-chain prefetching requires analysis to identify the pointer-chasing references, and to compute the work in the traversal code. Similar analyses are required for jump pointer techniques, though multi-chain prefetching must compute this information across multiple control structures. However, multi-chain prefetching does not require significant code modifications. While the true test of programming effort lies in compiler implementation, we believe multi-chain prefetching holds an advantage over jump pointer prefetching because it is far less intrusive.

6 Conclusion

This paper presents multi-chain prefetching, a novel latency tolerance technique that attacks the pointer chasing problem. Multi-chain prefetching performs prefetching using only the natural pointers in a linked data structure; however, it successfully hides serialized cache miss latency by aggressive prefetch scheduling to 1). exploit pre-loop overlap, and 2). overlap prefetches between multiple independent prefetch chains even though the prefetches within a single chain must perform serially.

Our results show that multi-chain prefetching is quite effective at tolerating serialized memory latency. On the four most memory-bound applications in our suite, multi-chain prefetching reduces overall execution time from 40% to 66%, and by 2.1% and 3.6% for the other two applications. We also show that multi-chain prefetching outperforms jump pointer prefetching from 24% to 64% for the memory-bound applications, and by 2.1% and 4.9% for the other two applications. The performance advantages of multi-chain prefetching come from its stateless nature. Multi-chain prefetching avoids the software overheads needed to create and manage jump pointer state. In addition, aggressive prefetch scheduling also results in higher cache miss coverage. Multi-chain prefetching can perform first-traversal prefetching, and can prefetch even the early nodes in a chain of pointers. Finally, we also believe multi-chain prefetching is easier to automate since it does not require the complex and intrusive code transformations associated with jump pointers.

The success of multi-chain prefetching relies on early initiation of prefetch chains to enable pre-loop overlap. This results in two limitations. First, many prefetches arrive too early, possibly evicting useful data prematurely. We show that the quality of prefetch timing can be improved if our scheduling algorithm obtains more accurate information to perform scheduling, an area for future work. Second, pointer chain addresses must be determined sufficiently early, otherwise pre-loop overlap may not be possible. In this paper, we assume static traversal patterns in which the prefetch addresses are known a priori. Jump pointer prefetching is similarly limited to static traversal patterns. Handling data-dependent traversal patterns is an ongoing area of research. Recently, prefetch arrays [8], an extension to jump pointer prefetching, was proposed to support a limited class of data-dependent traversals, so it can handle a wider range of pointer-chasing applications than our current multi-chain prefetching technique. In future research, we will investigate extensions to multi-chain prefetching to support data-dependent traversals.

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.

[2] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[3] Tien-Fu Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In *Proceedings of the 28th Annual Symposium on Microarchitecture*, pages 237–242. IEEE, 1995.

[4] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609–623, May 1995.

[5] Tzi cker Chiueh. Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops. In *In Proceedings of Supercomputing '94*, pages 488–497. ACM, November 1994.

[6] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *In Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, December 1992.

[7] Norman 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, Seattle, WA, May 1990. ACM.

[8] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, Toulouse, France, January 2000.

[9] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991. ACM.

[10] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996. ACM.

[11] Sharad Mehrotra and Luddy Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *In Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996. ACM.

[12] Todd Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems*, 16(1):55–92, February 1998.

[13] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[14] Subbarao Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, May 1994. ACM.

[15] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.

[16] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *In Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[17] Amir Roth and Gurindar S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[18] O. Temam. Streaming Prefetch. In *Proceedings of Europar'96*, Lyon,France, 1996.