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

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

Abstract

As the processor-memory performance gap continues to widen, application performance becomes increasingly limited by the memory system. Applications that employ linked data structures (LDSs) are particularly challenging from the standpoint of the memory system because of the memory serialization effects associated with indirect memory addressing. Also known as the *pointer chasing problem*, such memory serialization effects prevent latency tolerance techniques from overlapping the cache misses suffered along a chain of indirect memory references. Consequently, latency tolerance techniques are limited in their ability to tolerate the latency of cache misses arising from pointer chasing.

While pointer-chasing is inherently sequential, pointer-chasing computations typically traverse multiple pointer chains in an independent fashion. Such independent pointer-chasing traversals represent a large source of memory parallelism. This paper explores the possibility of exploiting such memory parallelism for the purposes of memory latency tolerance. First, we present a memory scheduling algorithm that computes a prefetch schedule from an LDS traversal specification that overlaps serialized memory fetches across multiple pointer-chasing traversals. Second, we present a prefetch engine architecture capable of traversing LDSs. Our prefetch engine issues prefetches according to the prefetch schedule, thus exploiting the memory parallelism uncovered by our scheduling algorithm. Finally, we conduct an experimental evaluation of our prefetching technique on four pointer-chasing applications. Our results show multi-chain prefetching increases performance between 52% and 78%. However, early initiation of prefetches causes prefetch buffer thrashing, thus limiting further increases in performance. Additional simulations show that prefetch buffer thrashing can be eliminated in some cases by using a larger prefetch buffer and by reducing the conservative prefetch distances computed by our scheduling algorithm.

1 Introduction

Prefetching has proven successful at hiding memory latency for applications that employ regular data structures (*e.g.* arrays). Unfortunately, conventional prefetching techniques are far less successful for pointer-intensive applications due to the memory serialization effects associated with linked data structure (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 [2, 8, 7, 5, 4]. These new techniques address the pointer-chasing problem using one of two different approaches. Techniques in the first approach [2, 8, 4], 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. Consequently, jump pointer techniques create memory parallelism for overlapping the cache misses suffered along a single chain of pointers. Techniques in the second approach [7, 5, 4], which we call *stateless techniques*, perform prefetching using only the natural pointers belonging to the LDS, and thus do not require additional state for prefetching. Existing stateless techniques prefetch pointer chains sequentially, and do not exploit any memory parallelism (or they exploit only limited forms of memory parallelism [7]). Instead, they schedule each prefetch as early in the loop iteration as possible to maximize the amount of work available for memory latency overlap.

Because of their ability to exploit memory parallelism, jump pointer techniques achieve higher levels of performance over a wider class of applications as compared to existing stateless techniques. 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, preliminary studies show that several important pointer-chasing applications consist mostly of traversal loops with small amounts of work [2]. However, jump pointer techniques are burdened with the creation and management of the prefetch state. The creation of jump pointers incurs runtime and memory overhead proportional to the size of the LDS. In addition, jump pointer techniques become less effective for applications with highly adaptive data structures since frequent inserts and deletes of link elements perturb the jump pointers. While the jump pointers can be modified as the LDS changes, this incurs additional runtime overhead. Finally, and perhaps most important, jump pointer techniques are intrusive. They require modification of the code's data structures, and insertion of jump pointer creation and management code. These transformations are complex and difficult to automate. Stateless techniques do not suffer from these drawbacks.

In this paper, we propose a new stateless prefetching technique called *multi-chain prefetching*. Like existing stateless techniques, our technique prefetches a single chain of pointers sequentially; however, we schedule 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. A scheduling algorithm ensures that prefetch chains are initiated sufficiently early to tolerate all the memory latency. As prefetch chains are scheduled increasingly early to accomodate long pointer chains, overlap occurs between multiple independent prefetch chains, thus creating memory parallelism across pointer chains that are traversed by separate loops or recursive function calls. Because such multi-chain overlap can be irregular, we rely on a prefetch engine to pursue independent pointer chains outside of the main CPU.

Multi-chain prefetching provides certain advantages over existing pointer prefetching techniques. Since multi-chain prefetching is stateless, it does not require code transformations to create and manage prefetch state as needed by jump pointer techniques, nor does it incur the runtime and memory overheads associated with the same. However, unlike existing stateless techniques, multi-chain prefetching more aggressively

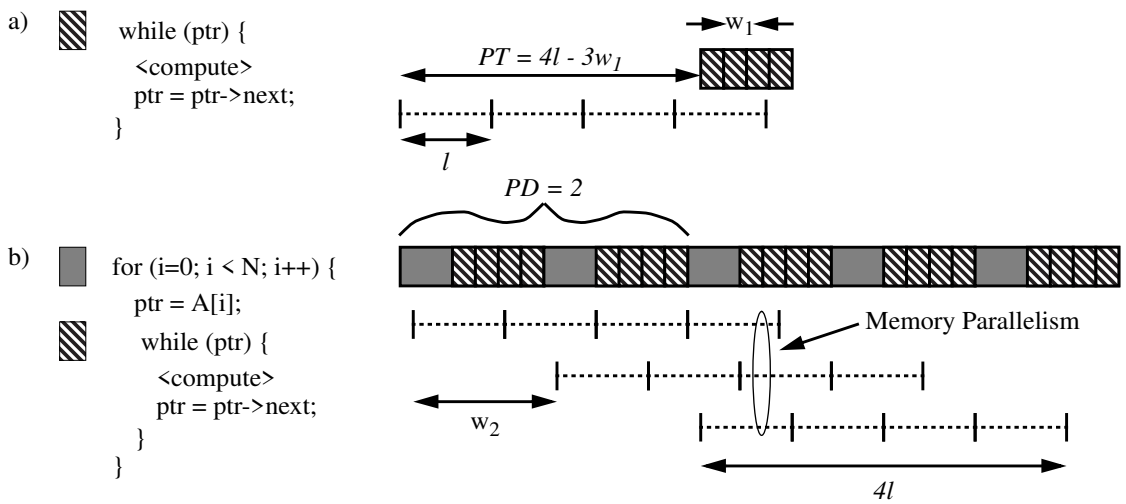

Figure 1: Overlapping pointer-chasing chains. a). Initiate prefetching of pointer chain prior to traversal loop. b). Exploit memory parallelism between independent pointer-chasing traversals.

schedules the prefetch of serialized memory latency. Consequently, multi-chain prefetching can effectively tolerate memory latency even when the traversal loops contain very little work.

The rest of this paper is organized as follows. Section 2 further explains the essence of our approach. Section 3 describes the scheduling algorithm to properly schedule the initiation of prefetch chains. Section 4 describes the prefetch engine support for our technique, and Section 5 presents experimental results. Finally, Section 6 presents the conclusions of the paper.

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 loop traverses a list of length four, performing some computation at each node in the list. The four hashed boxes in Figure 1a represent the processor's execution of the four loop iterations, each of which contains 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 issue serially and require $4l$ cycles to complete, 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 effectively 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), which guarantees that each prefetch arrives before its consuming loop iteration. 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 can tolerate the serialized memory latency for a single pointer chain, pointer chasing computations typically traverse many pointer chains. For example, repeated hash table lookups traverse a different linked list for each hash bucket, and in tree traversal, the traversal of the sub-trees at each node pursues independent chains. 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). In the Figure 1b example, $PD = 2$. Notice when $PD > 0$, the prefetches from separate chains overlap across iterations of the outer loop, hence 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 observes the code execution inside the processor and issues each chain in a synchronized fashion with the outer loop.

3 Prefetch Chain Scheduling

In this section, we present an off-line technique for scheduling prefetch chains, as described in Section 2. Our technique performs scheduling across multiple control structures since the overlap of cache misses in multi-chain prefetching spans multiple loops and/or recursive functions. We present our technique in two parts. First, in Section 3.1, we introduce an LDS traversal descriptor framework that captures the information required for prefetch chain scheduling. Our framework handles loops and recursive functions that traverse arrays, lists, trees, and the composition of all of these. Second, in Section 3.2, we describe a scheduling algorithm that identifies the asynchronous and synchronous prefetch traversals as discussed in Section 2, and computes a prefetch distance for each synchronous traversal.

3.1 LDS Descriptor Framework

To perform prefetch chain scheduling, we first extract information from the LDS traversal code necessary for scheduling. Two types of information are required: data structure layout, and traversal code work. We define a framework consisting of *LDS descriptors* to represent both the layout and work information. Figure 2 illustrates how our descriptors represent the data structure layout information, and Figure 3 shows the descriptor extensions that provide the traversal code work information. Our descriptors are also used by the prefetch engine to generate prefetch addresses at runtime, as discussed later in Section 4.

We specify data layout using two different descriptors, one for arrays and one for linked lists. As shown

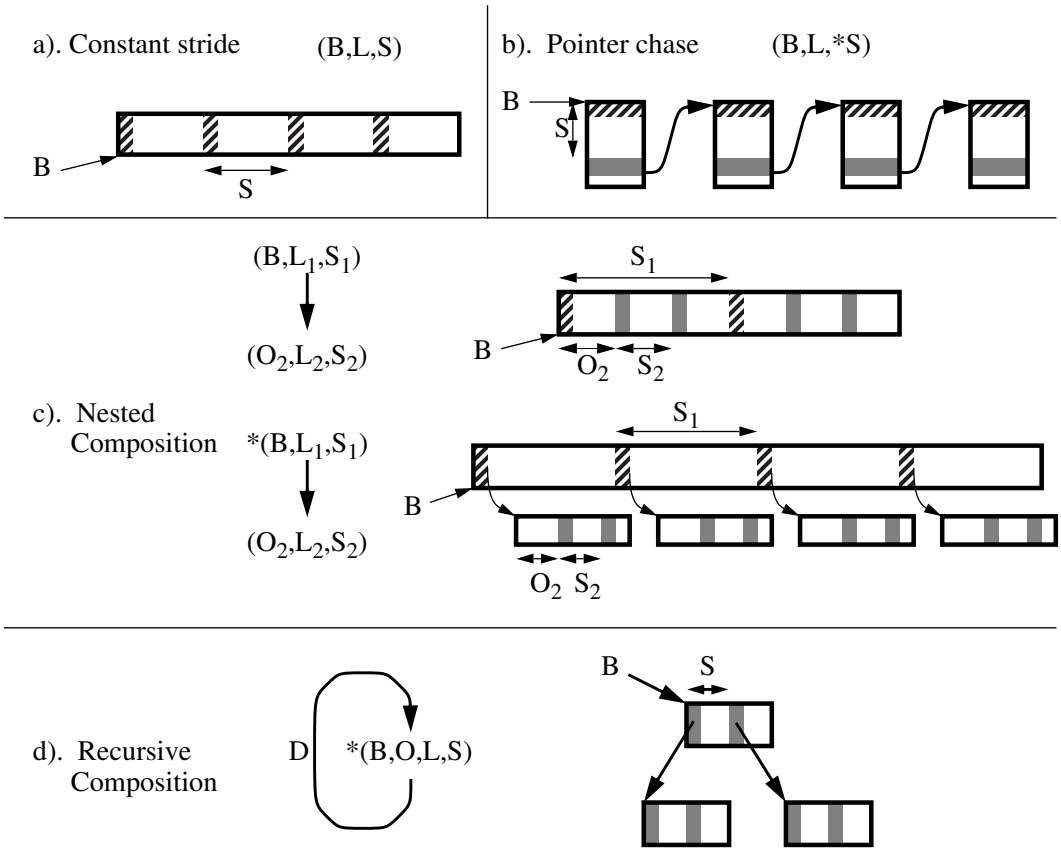

Figure 2: LDS descriptors: data layout information.

in Figure 2a, we represent an array structure with a descriptor that contains three parameters: *base* (B), *length* (L), and *stride* (S). The B parameter specifies the base address of the array, the L parameter specifies the number of array elements traversed by the application code, and the S parameter specifies the stride between consecutive references. This descriptor captures the layout of the array elements accessed by a loop that performs a constant-stride array traversal. As shown in Figure 2b, we represent a linked list structure with another descriptor that also contains three parameters. 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 enable representation of complex data structures, our framework permits descriptor composition. Composition is represented as a directed graph whose nodes are descriptors and whose edges are composition 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

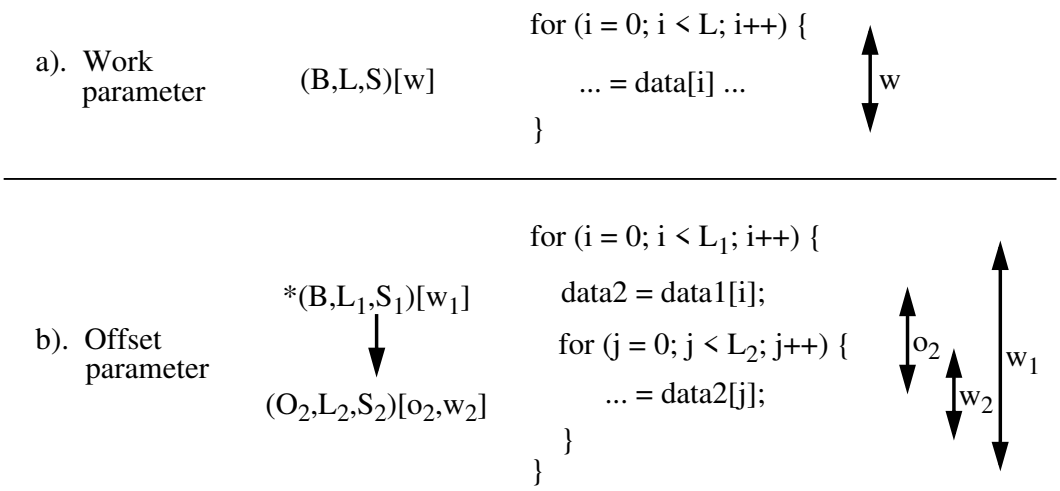

Figure 3: LDS descriptors: traversal code work information.

lower half of Figure 2c.[1] 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. Our framework also allows an arbitrary depth of nesting, though the prefetch engine may impose practical limitations.

In addition to nested composition, our framework permits recursive composition. Recursively composed descriptors behave identically to nested descriptors, except that the edge for the composition dependence is a backwards edge. 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 (in which case, both the B and O parameters are specified in the same descriptor), corresponding to a simple tree data structure.

Finally, Figure 3 shows the extensions to the descriptors in Figure 2 that provide the traversal code work information. The work information specifies the amount of work performed by the application code as it traverses the LDS specified by the data layout information. For each descriptor in the descriptor graph, two parameters are added. The *work* parameter, w, specifies the amount of work per traversal loop iteration. The work parameter is illustrated in Figure 3a. 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. The offset parameter is illustrated in Figure 3b.

3.2 Scheduling Algorithm

Once the descriptor information has been extracted from the LDS traversal code, we analyze the resulting descriptor graph and schedule all the prefetch chains. Two properties of the descriptor graph make scheduling

[1] These data structures have been referred to as "backbone-and-rib" structures [8].

```
for (i = N down to 1) {
    PTnest_i =  max  (PT_k - o_k)                          (1)
                composed k via indirection
    if ((descripor i is pointer-chasing) and (l > w_i)) {
        PT_i = L_i * (l - w_i) + w_i + PTnest_i            (2)
        PD_i = ∞;                                          (3)
    } else {
        PT_i = l + PTnest_i                                (4)
        PD_i = ⌈PT_i / w_i⌉                                (5)
    }
}
```

Figure 4: Scheduling algorithm for non-cyclic static descriptor graphs.

challenging: cycles due to recursively composed descriptors, and descriptor parameters that are not known until runtime (*i.e.* dynamic descriptor graphs). In this section, we first describe how scheduling is performed for acyclic static descriptor graphs. Then, we briefly discuss extensions to handle both cyclic and dynamic descriptor graphs.

3.2.1 Acyclic Static Descriptor Graphs

Our scheduling algorithm computes three scheduling parameters for each descriptor in the descriptor graph: whether the descriptor requires asynchronous or synchronous prefetching, the pre-loop time, PT, and if synchronous prefetching is to be used, the prefetch distance, PD (see Section 2 for a discussion of these parameters). Figure 4 presents a scheduling algorithm that computes these parameters. Since our scheduling algorithm is defined recursively, we must process the descriptors beginning at the leaves of the descriptor graph and work towards the top. Assuming there are N descriptors in the descriptor graph, and assuming we assign a number to each descriptor between 1 and N in top-down order, then the "for (N down to 1)" loop in Figure 4 visits the descriptors in the required bottom-up order. 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 $l <= w_i$, then it requires synchronous prefetching.[2] The "if" conditional test in Figure 4 computes whether asynchronous or synchronous prefetching is used.

Next, we compute the pre-loop time, PT. For asynchronous prefetching, we must exploit work prior to the traversal loop to hide the excess serialized memory latency that cannot be hidden underneath the traversal loop itself. From Figure 1, we saw that $PT = 4l - 3w_1$ for an 4-iteration pointer-chasing loop. In

[2]Notice 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 itself.

general, $PT_i = L_i * (l - w_i) + w_i$. For synchronous prefetching, we need to hide the cache miss for the data consumed only by the first iteration of the traversal loop, so $PT_i = l$. These equations for PT conservatively assume every prefetch will miss in the cache and suffer the full l-cycle latency to memory because determining which prefetches will actually miss in the cache is difficult, particularly for pointer-chasing traversals.

Equations 2 and 4 in Figure 4 compute the PT parameter 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 the base address for the inner descriptor k cannot be computed until each corresponding address in the outer descriptor i is computed. 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. 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.

3.2.2 Handling Cyclic Dynamic Descriptor Graphs

Section 3.2.1 describes our scheduling algorithm assuming the descriptor graph is acyclic. In general, descriptor graphs are cyclic due to recursive descriptor composition. We preprocess cyclic graphs to convert them into acyclic graphs, and then use the same scheduling algorithm described in Section 3.2.1 to compute the scheduling parameters. The preprocessing step converts each recursive composition in the descriptor graph into a nested composition by replicating all the descriptors involved in the cycle D times, nesting each replicated copy underneath the next. Children nodes of the replicated descriptors that are not involved in the cycle need not be replicated; however, the dependence edges leading to such children are replicated once for each new descriptor created.

In addition to having cycles, descriptor graphs are typically dynamic as well–they contain parameters that are unknown at compile time such as list length and recursion depth. 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 insight 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. These statically computed prefetch distances can then be used at runtime for prefetching.

Because they represent an upper bound, the bounded prefetch distances guarantee that all prefetches are initiated far enough in advance to tolerate their memory latencies without using any runtime information. However, the bounds can be conservative; hence, using the bounded values for prefetching may initiate prefetches earlier than necessary. In Section 5, we will quantify this effect.

4 Prefetch Engine

In multi-chain prefetching, prefetch requests along a chain of pointers issue asynchronously with respect to the traversal loop (see Figure 1); therefore, it is difficult or impossible to software pipeline the prefetches into the loop code. In this section, we introduce a programmable prefetch engine architecture that dynamically issues the prefetches according to our scheduling algorithm outside of the main CPU. Section 4.1 describes how our prefetch engine is integrated with a commodity CPU, Section 4.2 describes the address generation hardware, and Section 4.3 discusses how prefetches are scheduled.

4.1 Integration with a CPU

The hardware organization of the prefetch engine appears in Figure 5. The design requires three additions to a commodity microprocessor: a prefetch buffer, the prefetch engine itself, and two new instructions called $SINIT$ and $SYNC$.

Our prefetch buffer is similar to the prefetch buffers proposed for software prefetching [3]. It is a fully associative buffer that stages prefetched data prior to its access by the processor. Each time the prefetch engine issues a new prefetch request, the request address is checked in the processor's cache and in the prefetch buffer. If a miss occurs in both places, a new entry in the prefetch buffer is allocated and a memory request is initiated for the cache block that missed. When the request completes, the cache block is loaded into the prefetch buffer. All memory operations performed by the processor check both the cache and the prefetch buffer. If a hit in the prefetch buffer occurs, the corresponding cache block is moved into the cache.

In the next section, we discuss the prefetch engine internals and describe how the ISA extensions are used to help properly schedule prefetch requests.

4.2 Address Generation Hardware

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 node 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 node number within

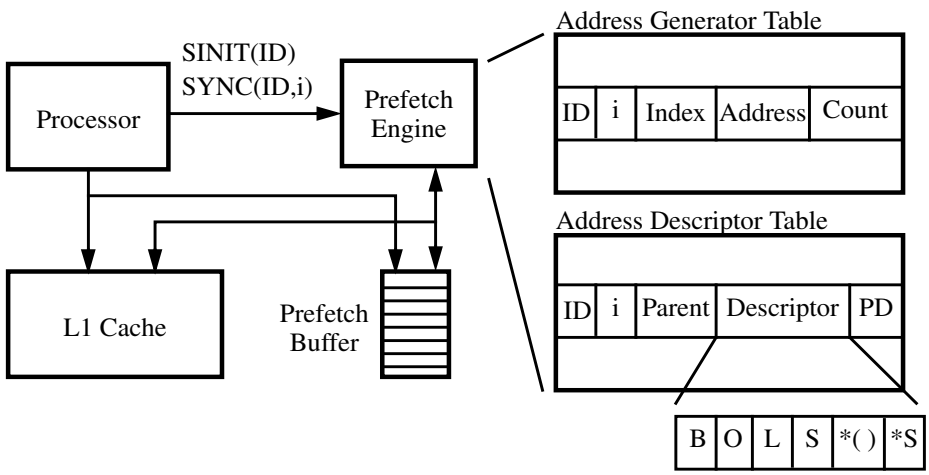

Figure 5: Prefetch engine hardware.

the graph, i, assuming the top-down numbering of nodes discussed in Section 3.2.1. The *Parent* field in ADT entry i specifies the number of descriptor i's parent in the descriptor graph. The next six values, B, O, L, S, $*()$, and $*S$, record the descriptor parameters that specify the data layout information. Here, $*S$ is a 1-bit value that specifies constant-stride versus pointer-chasing address generation, and $*()$ is another 1-bit value that specifies the use of indirection between composed descriptors. 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 provides dynamic storage for address computation during prefetching. AGT entries are activated dynamically as prefetch addresses are generated. Once 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 the descriptor graph ID. The prefetch engine searches the ADT for the entry with the matching ID and $i = 0$ (*i.e.* the root node of the descriptor graph). 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, 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, address generation terminates, and the AGT entry is deactivated.

4.3 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 data structure described by descriptor i. 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 corresponding 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 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 present experimental results that evaluate the performance of our approach. Section 5.1 describes the methodology used for our experiments, Section 5.2 presents the baseline performance results, and Section 5.3 discusses techniques to handle prefetches that are initiated too early.

Application	Data Structure	Input Parameters
EM3D	array of pointers	10,000 nodes
MST	list of lists	1024 nodes
Treeadd	binary tree	20 levels
Health	quadtree of lists	5 levels, 500 iters

Table 1: Benchmark summary.

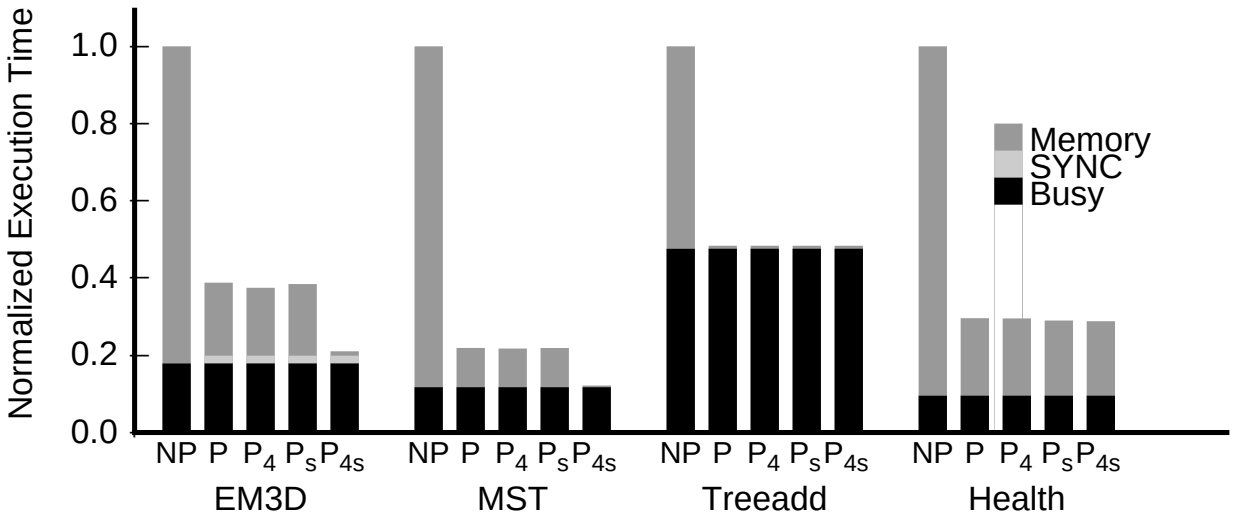

Figure 6: Execution time for no prefetching (NP) and four versions of multi-chain prefetching (P, P_4, P_s, and P_{4s}). Each execution time bar has been broken down into useful cycles (Busy), cycles spent executing SYNC and SINIT instructions (SYNC), and memory stall cycles (Memory).

5.1 Experimental Methodology

To evaluate the performance of our multi-chain prefetching technique, 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. The L1 cache is a split 16-Kbyte instruction/16-Kbyte data direct-mapped write-through cache with a 32-byte block size. The L2 cache is a unified 512-Kbyte 4-way set-associative write-back cache with 64-byte cache blocks. We assume the L1 cache has 8 MSHRs and the L2 cache has 16 MSHRs to enable significant memory concurrency. We also allow the memory sub-system to aggressively support a peak bandwidth of 8-Gbytes/sec. The prefetch engine is integrated on the same chip as the processor and clocks at the same rate. We assume 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.

Table 1 summarizes the four applications used to evaluate our approach–EM3D, MST, Treeadd, and Health. All four applications come from the Olden benchmark suite [6]. In Table 1, the primary data

structures found in the applications, as well as the input parameters used in our simulations are listed.

5.2 Prefetching Performance

Figure 6 presents the results of multi-chain prefetching on our four applications. For each application, five bars show the execution time breakdown using various forms of our prefetching technique (the bars are explained below). Each execution time bar has been broken down into three components: time spent executing useful instructions, time spent executing $SYNC$ and $SINIT$ instructions, and time spent in memory stall, labeled "Busy," "SYNC," and "Memory," respectively. All times have been normalized against the leftmost bar for each application, labeled "NP."

The execution bars labeled "NP" show the performance of the applications without prefetching. As these bars indicate, our applications are severely memory bound, spending anywhere between 52% and 91% stalled on the memory system. These large memory overheads exist despite the efforts of out-of-order execution to overlap the cache miss latencies. Out-of-order execution is less effective at tolerating the memory latency of pointer-chasing references due to the serialized nature of the cache misses along a chain of pointers.

The "P" bars report the running time of the applications with multi-chain prefetching. Multi-chain prefetching successfully overlaps most of the exposed memory latency for all four applications despite the serialized nature of pointer-chasing memory references by exploiting memory parallelism across independent pointer chains. A comparison of the "P" bars against the "NP" bars shows that our prefetch engine reduces overall runtime between 52% and 78%. Note that the overhead of the $SYNC$ and $SINIT$ instructions are negligible for all four applications; hence, practically all of the reduction in memory stall achieved by the prefetch engine directly translates to improved execution time.

While multi-chain prefetching eliminates a large fraction of the memory stall, a significant memory stall component still exists for EM3D, MST, and Health. A closer look at these applications revealed that the remaining stalls are due to the early nature in which multi-chain prefetching initiates prefetches to overlap serialized memory latency with pre-loop work. Prefetches that arrive in advance of when they are consumed by the processor are placed in the prefetch buffer. For applications with long pointer chains (or alternatively, short traversal loops), the number of early prefetches increases. When the number of early prefetches exceeds the prefetch buffer size, the prefetches evict each other before they are consumed by the processor. Note that prefetched data is placed in the L2 cache as well as the prefetch buffer; therefore, even if a prefetch is evicted from the prefetch buffer before its use, the processor will normally enjoy a hit to the L2 cache where prefetch thrashing is avoided due to the much larger size of the L2.

5.3 Handling Early Prefetches

We conducted several experiments to see whether the early prefetch problem inherent in multi-chain prefetching can be addressed. First, we increased the size of the prefetch buffer from 1K to 4K. The bars labeled "P_4"

in Figure 6 show the impact of this enhancement. Except for EM3D which experiences a slight performance increase, there is no perceptible change due to an increased prefetch buffer size.

Second, we reduced the prefetch distance computed by our scheduling algorithm. Our scheduling algorithm is overly conservative for two reasons. First, the current algorithm does not perform any locality analysis, and assumes that each scheduled prefetch will miss all the way to main memory. In practice, a large fraction of prefetches will find the desired data in the L1 or L2 cache due to undetected locality. Second, our scheduling algorithm computes the upper bound on prefetch distance since it does not have runtime information on the extent of dynamic data structures. Both of these factors cause our scheduling algorithm to compute prefetch distances that are too large. In the bars labeled "P_s" and "P_{4s}", we report execution times when all prefetch distances are shortened by a factor of two for the 1K and 4K prefetch buffer sizes, respectively.

The "P_s" bars show that shortening the prefetch distance alone does not help, though an important result is that it does not hurt performance for any of the applications indicating that the original prefetch distances are indeed overly conservative. However, when the shortened prefetch distances are combined with the larger prefetch buffer size (the "P_{4s}" bars), the number of early prefetches is reduced enough to eliminate prefetch buffer thrashing in EM3D and MST, allowing most of the remaining stalls to be overlapped. Unfortunately, none of our efforts could address the early prefetches in Health. Health traverses linked lists of length 150 or greater. Multi-chain prefetching requires significant buffering to overlap the misses for this application, and so it is only effective at prefetching into the L2 cache.

6 Conclusion

While pointer-chasing is inherently sequential, pointer-chasing applications typically traverse multiple independent pointer chains. Such independent traversals represent a large source of memory parallelism that remains untapped by existing memory latency tolerance techniques. This paper explores a new prefetching technique, called multi-chain prefetching, that exploits the memory parallelism across independent pointer-chasing traversals. We present a memory scheduling algorithm that aggressively overlaps prefetches from independent pointer-chasing traversals to eliminate memory stalls. In addition, we present a prefetch engine architecture capable of traversing LDSs and issuing prefetches according to the prefetch schedule, thus exploiting the memory parallelism uncovered by our scheduling algorithm.

To evaluate multi-chain prefetching, we conduct a simulation-based study using four pointer-intensive applications. Our early results are encouraging, demonstrating performance increases between 52% and 78%. The performance evaluation also reveals that early initiation of prefetches causes prefetch buffer thrashing, thus limiting further increases in performance. The primary cause for the early prefetches is the overly conservative nature in which our off-line scheduling algorithm computes prefetch distances due to the lack of locality and data structure extent information. Our results show that when the prefetch distance computed

by our scheduling algorithm is reduced by a factor of 2 and the prefetch buffer size is increased to 4K, prefetch buffer thrashing is eliminated for two of our applications. In future work, we will investigate locality analysis and runtime profiling to provide our scheduling algorithm with more accurate information.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.