

An earlier version of this work also appears in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques. September 2001*.

## A General Framework for Prefetch Scheduling in Linked Data Structures and its Application to Multi-Chain Prefetching

Seungryul Choi<sup>‡</sup>, Nicholas Kohout<sup>††</sup>, Sumit Pamnani<sup>\*</sup>, Dongkeun Kim<sup>†</sup>, Donald Yeung<sup>†</sup>

<sup>‡</sup>Computer Science Dept.  
Univ. of Maryland,  
College Park  
choi@cs.umd.edu

<sup>††</sup>Intel Corp.  
nicholas.j.kohout  
@intel.com

<sup>\*</sup>AMD Inc.  
sumitkumar.pamnani  
@amd.com

<sup>†</sup>Electrical & Computer Eng. Dept.  
Univ. of Maryland,  
College Park  
{dongkeun,yeung}@eng.umd.edu

### Abstract

Pointer-chasing applications tend to traverse composite data structures consisting of multiple independent pointer chains. While the traversal of any single pointer chain leads to the serialization of memory operations, the traversal of independent pointer chains provides a source of memory parallelism. This article investigates exploiting such *inter-chain memory parallelism* for the purpose of memory latency tolerance, using a technique called *multi-chain prefetching*. Previous works [30, 31] have proposed prefetching simple pointer-based structures in a multi-chain fashion. However, our work enables multi-chain prefetching for arbitrary data structures composed of lists, trees, and arrays.

This article makes five contributions in the context of multi-chain prefetching. First, we introduce a framework for compactly describing LDS traversals, providing the data layout and traversal code work information necessary for prefetching. Second, we present an off-line scheduling algorithm for computing a prefetch schedule from the LDS descriptors that overlaps serialized cache misses across separate pointer-chain traversals. Our analysis focuses on static traversals. We also propose using speculation to identify independent pointer chains in dynamic traversals. Third, we propose a hardware prefetch engine that traverses pointer-based data structures and overlaps multiple pointer chains according to the computed prefetch schedule. Fourth, we present a compiler that extracts LDS descriptors via static analysis of the application source code, thus automating multi-chain prefetching. Finally, we conduct an experimental evaluation of compiler-instrumented multi-chain prefetching and compare it against jump pointer prefetching [21], prefetch arrays [14], and predictor-directed stream buffers (PSB) [33].

Our results show compiler-instrumented multi-chain prefetching improves execution time by 40% across six pointer-chasing kernels from the Olden benchmark suite [29], and by 3% across four SPECint2000 benchmarks. Compared to jump pointer prefetching and prefetch arrays, multi-chain prefetching achieves 34% and 11% higher performance for the selected Olden and SPECint2000 benchmarks, respectively. Compared to PSB, multi-chain prefetching achieves 27% higher performance for the selected Olden benchmarks, but PSB outperforms multi-chain prefetching by 0.2% for the selected SPECint2000 benchmarks. An ideal PSB with an infinite markov predictor achieves comparable performance to multi-chain prefetching, coming within 6% across all benchmarks. Finally, speculation can enable multi-chain prefetching for some dynamic traversal codes, but our technique loses its effectiveness when the pointer-chain traversal order is highly dynamic.

# 1 Introduction

A growing number of important non-numeric applications employ linked data structures (LDSs). For example, databases commonly use index trees and hash tables to provide quick access to large amounts of data. Several compression algorithms and voice recognition programs use hash tables to store lookup values. Finally, object-oriented programming environments such as C++ and Java track dynamic objects using object graphs and invoke methods via function tables, both requiring linked data structures. Moreover, current trends show these non-numeric codes will become increasingly important relative to scientific workloads on future high-performance platforms.

The use of LDSs will likely have a negative impact on memory performance, making many non-numeric applications severely memory-bound on future systems. LDSs can be very large owing to their dynamic heap construction. Consequently, the working sets of codes that use LDSs can easily grow too large to fit in the processor’s cache. In addition, logically adjacent nodes in an LDS may not reside physically close in memory. As a result, traversal of an LDS may lack spatial locality, and thus may not benefit from large cache blocks. The sparse memory access nature of LDS traversal also reduces the effective size of the cache, further increasing cache misses.

In the past, researchers have used prefetching to address the performance bottlenecks of memory-bound applications. Several techniques have been proposed, including software prefetching techniques [2, 16, 25, 26], hardware prefetching techniques [5, 9, 13, 28], or hybrid techniques [4, 6, 35]. While such conventional prefetching techniques are highly effective for applications that employ regular data structures (*e.g.* arrays), these techniques are far less successful for non-numeric applications that make heavy use of LDSs due to memory serialization effects 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* during LDS traversal prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain.

Recently, researchers have begun investigating prefetching techniques designed for LDS traversals. These new LDS prefetching techniques address the pointer-chasing problem using several different approaches. *Stateless techniques* [21, 23, 30, 37] prefetch pointer chains sequentially using only the natural pointers belonging to the LDS. Existing stateless techniques do not exploit any memory parallelism at all, or they exploit only limited amounts of memory parallelism. Consequently, they lose their effectiveness when the LDS traversal code contains insufficient work to hide the serialized memory latency [21].

A second approach [14, 21, 31], which we call *jump pointer techniques*, inserts 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. Because they create memory parallelism using jump pointers, jump pointer techniques tolerate pointer-chasing cache misses even when the traversal loops contain insufficient work to hide the serialized memory latency. However, jump pointer techniques cannot commence prefetching until the jump pointers have been installed. Furthermore, the jump pointer installation code increases execution time, and the jump pointers themselves contribute additional cache misses.

Finally, a third approach consists of *prediction-based techniques* [12, 33, 34]. These techniques perform prefetching by predicting the cache-miss address stream, for example using hardware predictors [12, 33]. Early hardware predictors were capable of following striding streams only, but

more recently, correlation [3] and markov [12] predictors have been proposed that can follow arbitrary streams, thus enabling prefetching for LDS traversals. Because predictors need not traverse program data structures to generate the prefetch addresses, they avoid the pointer-chasing problem altogether. In addition, for hardware prediction, the techniques are completely transparent since they require no support from the programmer or compiler. However, prediction-based techniques lose their effectiveness when the cache-miss address stream is unpredictable.

This article investigates exploiting the natural memory parallelism that exists between independent serialized pointer-chasing traversals, or *inter-chain memory parallelism*. Our approach, called *multi-chain prefetching*, issues prefetches along a single chain of pointers sequentially, but aggressively pursues multiple independent pointer chains simultaneously whenever possible. Due to its aggressive exploitation of inter-chain memory parallelism, multi-chain prefetching can tolerate serialized memory latency even when LDS traversal loops have very little work; hence, it can achieve higher performance than previous stateless techniques. Furthermore, multi-chain prefetching does not use jump pointers. As a result, it does not suffer the overheads associated with creating and managing jump pointer state. And finally, multi-chain prefetching is an execution-based technique, so it is effective even for programs that exhibit unpredictable cache-miss address streams.

The idea of overlapping chained prefetches, which is fundamental to multi-chain prefetching, is not new: both Cooperative Chain Jumping [31] and Dependence-Based Prefetching [30] already demonstrate that simple “backbone and rib” structures can be prefetched in a multi-chain fashion. However, our work pushes this basic idea to its logical limit, enabling multi-chain prefetching for arbitrary data structures (our approach can exploit inter-chain memory parallelism for any data structure composed of lists, trees, and arrays). Furthermore, previous chained prefetching techniques issue prefetches in a greedy fashion. In contrast, our work provides a formal and systematic method for scheduling prefetches that controls the timing of chained prefetches. By controlling prefetch arrival, multi-chain prefetching can reduce both early and late prefetches which degrade performance compared to previous chained prefetching techniques.

In this article, we build upon our original work in multi-chain prefetching [17], and make the following contributions:

1. We present an *LDS descriptor framework* for specifying static LDS traversals in a compact fashion. Our LDS descriptors contain data layout information and traversal code work information necessary for prefetching.
2. We develop an off-line algorithm for computing an exact prefetch schedule from the LDS descriptors that overlaps serialized cache misses across separate pointer-chain traversals. Our algorithm handles static LDS traversals involving either loops or recursion. Furthermore, our algorithm computes a schedule even when the extent of dynamic data structures is unknown. To handle dynamic LDS traversals, we propose using speculation. However, our technique cannot handle codes in which the pointer-chain traversals are highly dynamic.
3. We present the design of a programmable prefetch engine that performs LDS traversal outside of the main CPU, and prefetches the LDS data using our LDS descriptors and the prefetch schedule computed by our scheduling algorithm. We also perform a detailed analysis of the hardware cost of our prefetch engine.
4. We introduce algorithms for extracting LDS descriptors from application source code via static analysis, and implement them in a prototype compiler using the SUIF framework [11].

Our prototype compiler is capable of extracting all the program-level information necessary for multi-chain prefetching fully automatically.

5. Finally, we conduct an experimental evaluation of multi-chain prefetching using several pointer-intensive applications. Our evaluation compares compiler-instrumented multi-chain prefetching against jump pointer prefetching [21, 31] and prefetch arrays [14], two jump pointer techniques, as well as predictor-directed stream buffers [33], an all-hardware prediction-based technique. We also investigate the impact of early prefetch arrival on prefetching performance, and we compare compiler- and manually-instrumented multi-chain prefetching to evaluate the quality of the instrumentation generated by our compiler. In addition, we characterize the sensitivity of our technique to varying hardware parameters. Lastly, we undertake a preliminary evaluation of *speculative multi-chain prefetching* to demonstrate its potential in enabling multi-chain prefetching for dynamic LDS traversals.

The rest of this article is organized as follows. Section 2 further explains the essence of multi-chain prefetching. Then, Section 3 introduces our LDS descriptor framework. Next, Section 4 describes our scheduling algorithm, Section 5 discusses our prefetch engine, and Section 6 presents our compiler for automating multi-chain prefetching. After presenting all our algorithms and techniques, Sections 7 and 8 then report on our experimental methodology and evaluation, respectively. Finally, Section 9 discusses related work, and Section 10 concludes the article.

## 2 Multi-Chain Prefetching

This section provides an overview of our multi-chain prefetching technique. Section 2.1 presents the idea of exploiting inter-chain memory parallelism. Then, Section 2.2 discusses the identification of independent pointer chain traversals.

### 2.1 Exploiting Inter-Chain Memory Parallelism

The multi-chain prefetching technique augments a commodity microprocessor with a programmable hardware prefetch engine. During an LDS computation, the prefetch engine performs its own traversal of the LDS in front of the processor, thus prefetching the LDS data. The prefetch engine, however, is capable of traversing multiple pointer chains simultaneously when permitted by the application. Consequently, the prefetch engine can tolerate serialized memory latency by overlapping cache misses across independent pointer-chain traversals.

To illustrate the idea of exploiting *inter-chain memory parallelism*, we first describe how our prefetch engine traverses a single chain of pointers. Figure 1a shows a loop that traverses a linked list of length three. Each loop iteration, denoted by a hashed box, contains  $w_1$  cycles of work. Before entering the loop, the processor executes a prefetch directive,  $INIT(ID_U)$ , instructing the prefetch engine to initiate traversal of the linked list identified by the  $ID_U$  label. If all three link nodes suffer an  $l$ -cycle cache miss, the linked list traversal requires  $3l$  cycles since the link nodes must be fetched sequentially. Assuming  $l > w_1$ , the loop alone contains insufficient work to hide the serialized memory latency. As a result, the processor stalls for  $3l - 2w_1$  cycles. To hide these stalls, the prefetch engine would have to initiate its linked list traversal  $3l - 2w_1$  cycles before the processor traversal. For this reason, we call this delay the *pre-traversal time (PT)*.

While a single pointer chain traversal does not provide much opportunity for latency tolerance,

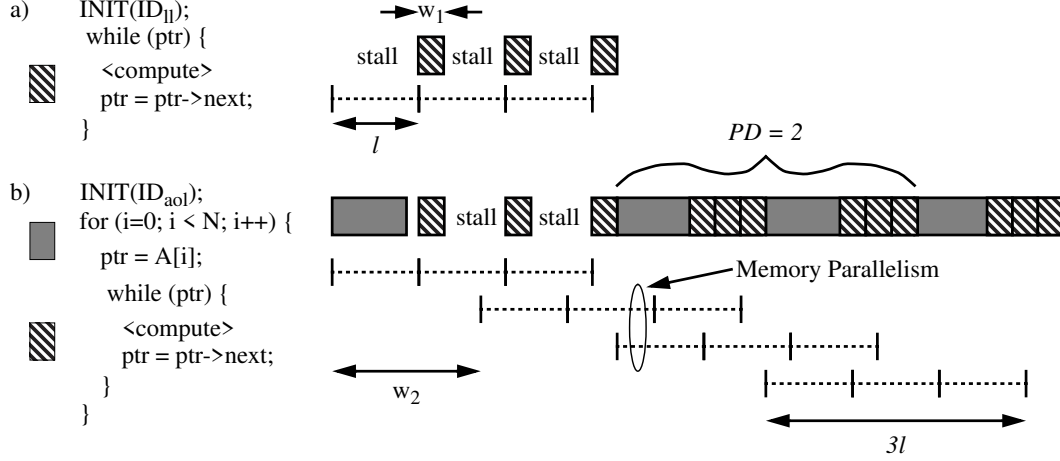


Figure 1: Traversing pointer chains using a prefetch engine. a). Traversal of a single linked list. b). Traversal of an array of lists data structure.

pointer chasing computations typically traverse many pointer chains, each of which is often independent. To illustrate how our prefetch engine exploits such independent pointer-chasing traversals, Figure 1b shows a doubly nested loop that traverses an array of lists data structure. The outer loop, denoted by a shaded box with  $w_2$  cycles of work, traverses an array that extracts a head pointer for the inner loop. The inner loop is identical to the loop in Figure 1a.

In Figure 1b, the processor again executes a prefetch directive,  $INIT(ID_{aol})$ , causing the prefetch engine to initiate a traversal of the array of lists data structure identified by the  $ID_{aol}$  label. As in Figure 1a, the first linked list is traversed sequentially, and the processor stalls since there is insufficient work to hide the serialized cache misses. However, the prefetch engine then initiates the traversal of subsequent linked lists in a pipelined fashion. If the prefetch engine starts a new traversal every  $w_2$  cycles, then each linked list traversal will initiate the required  $PT$  cycles in advance, thus hiding the excess serialized memory latency across multiple outer loop iterations. The number of outer loop iterations required to overlap each linked list traversal is called the *prefetch distance* ( $PD$ ). Notice when  $PD > 1$ , the traversals of separate chains overlap, exposing inter-chain memory parallelism despite the fact that each chain is fetched serially.

## 2.2 Finding Independent Pointer-Chain Traversals

In order to exploit inter-chain memory parallelism, it is necessary to identify multiple independent pointer chains so that our prefetch engine can traverse them in parallel and overlap their cache misses, as illustrated in Figure 1. An important question is whether such independent pointer-chain traversals can be easily identified.

Many applications perform traversals of linked data structures in which the order of link node traversal does not depend on runtime data. We call these *static traversals*. The traversal order of link nodes in a static traversal can be determined a priori via analysis of the code, thus identifying the independent pointer-chain traversals at compile time. In this paper, we present an LDS descriptor framework that compactly expresses the LDS traversal order for static traversals. The descriptors in our framework also contain the data layout information used by our prefetch engine to generate the sequence of load and prefetch addresses necessary to perform the LDS traversal at runtime.

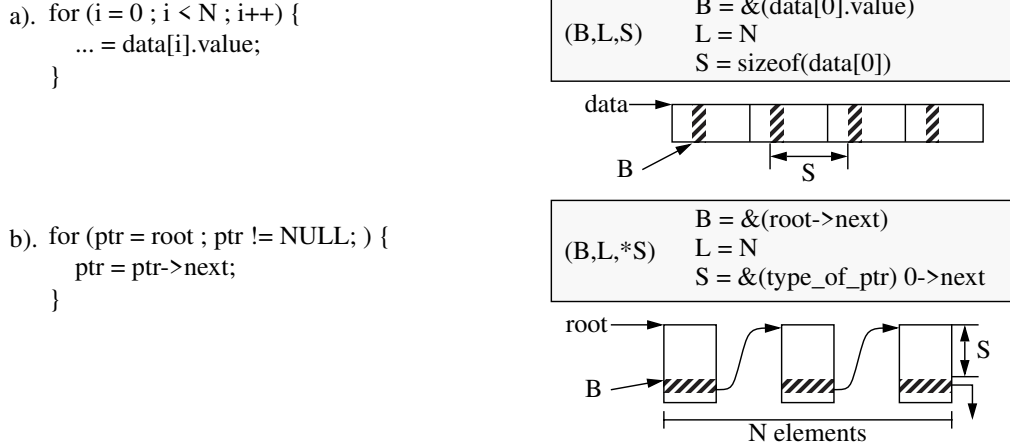


Figure 2: Two LDS descriptors used to specify data layout information. a). Array descriptor. b). Linked list descriptor. Each descriptor appears inside a box, and is accompanied by a traversal code example and an illustration of the data structure.

While compile-time analysis of the code can identify independent pointer chains for static traversals, the same approach does not work for *dynamic traversals*. In dynamic traversals, the order of pointer-chain traversal is determined at runtime. Consequently, the simultaneous prefetching of independent pointer chains is limited since the chains to prefetch are not known until the traversal order is computed, which may be too late to enable inter-chain overlap. For dynamic traversals, it may be possible to *speculate* the order of pointer-chain traversal if the order is predictable. In this paper, we focus on static LDS traversals. Later in Section 8.7, we illustrate the potential for predicting pointer-chain traversal order in dynamic LDS traversals by extending our basic multi-chain prefetching technique with speculation.

### 3 LDS Descriptor Framework

Having provided an overview of multi-chain prefetching, we now explore the algorithms and hardware underlying its implementation. We begin by introducing a general framework for compactly representing static LDS traversals, which we call the *LDS descriptor framework*. This framework allows compilers (and programmers) to compactly specify two types of information related to LDS traversal: data structure layout, and traversal code work. The former captures memory reference dependences that occur in an LDS traversal, thus identifying pointer-chasing chains, while the latter quantifies the amount of computation performed as an LDS is traversed. After presenting the LDS descriptor framework, subsequent sections of this article will show how the information provided by the framework is used to perform multi-chain prefetching (Sections 4 and 5), and how the LDS descriptors themselves can be extracted by a compiler (Section 6).

#### 3.1 Data Structure Layout Information

Data structure layout is specified using two descriptors, one for arrays and one for linked lists. Figure 2 presents each descriptor along with a traversal code example and an illustration of the traversed data structure. The array descriptor, shown in Figure 2a, contains three parameters: base

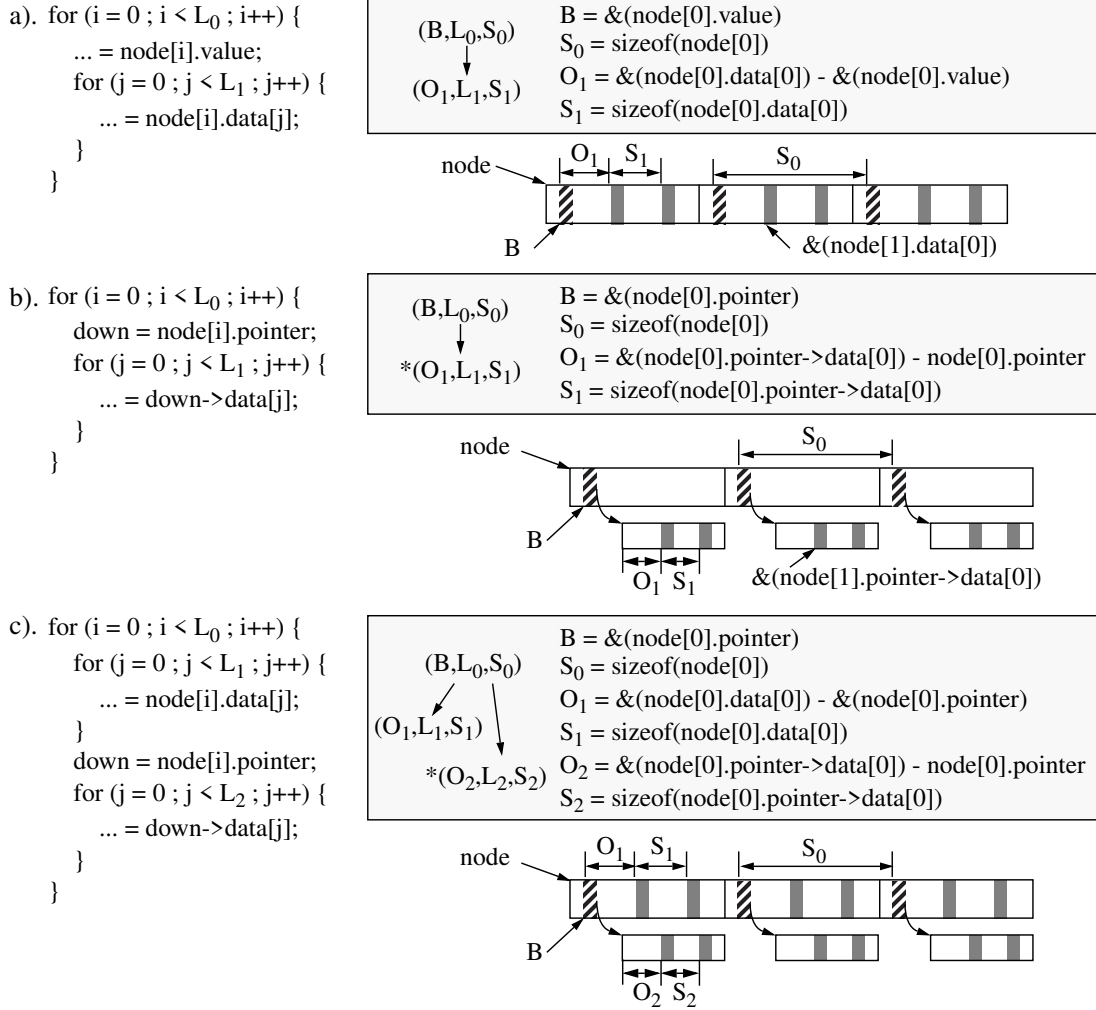


Figure 3: Nested descriptor composition. a). Nesting without indirection. b). Nesting with indirection. c). Nesting multiple descriptors. Each descriptor composition appears inside a box, and is accompanied by a traversal code example and an illustration of the composite data structure.

( $B$ ), 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. The array descriptor specifies the memory address stream emitted by the processor during 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. The linked list descriptor specifies the memory address stream emitted by the processor during a linked list traversal.

To specify the layout of complex data structures, our framework permits descriptor composition. Descriptor 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. The first type of composition is *nested composition*. In nested composition, each

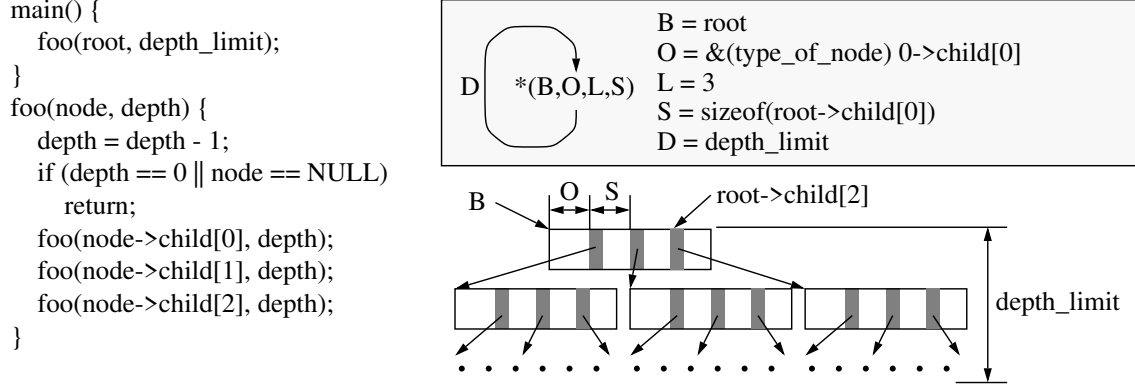


Figure 4: Recursive descriptor composition. The recursive descriptor appears inside a box, and is accompanied by a traversal code example and an illustration of the tree data structure.

address generated by an outer descriptor forms the  $B$  parameter for multiple instantiations of a dependent inner descriptor. An offset parameter,  $O$ , is specified in place of the inner descriptor’s  $B$  parameter to shift its base address by a constant offset. Such nested descriptors capture the memory reference streams of nested loops that traverse multi-dimensional data structures. Figure 3 presents several nested descriptors, showing a traversal code example and an illustration of the traversed multi-dimensional data structure along with each nested descriptor.

Figure 3a shows the traversal of an array of structures, each structure itself containing an array. The code example’s outer loop traverses the array “node,” accessing the field “value” from each traversed structure, and the inner loop traverses each embedded array “data.” The outer and inner array descriptors,  $(B, L_0, S_0)$  and  $(O_1, L_1, S_1)$ , represent the address streams produced by the outer and inner loop traversals, respectively. (In the inner descriptor, “ $O_1$ ” specifies the offset of each inner array from the top of each structure). Figure 3b illustrates another form of descriptor nesting in which indirection is used between nested descriptors. The data structure in Figure 3b is similar to the one in Figure 3a, except the inner arrays are allocated separately, and a field from each outer array structure, “node[i].pointer,” points to a structure containing the inner array. Hence, as shown in the code example from Figure 3b, traversal of the inner array requires indirection through the outer array’s pointer to compute the inner array’s base address. In our framework, this indirection is denoted by placing a “\*” in front of the inner descriptor. Figure 3c, our last nested descriptor example, illustrates the nesting of multiple inner descriptors underneath a single outer descriptor to represent the address stream produced by nested distributed loops. The code example from Figure 3c shows the two inner loops from Figures 3a-b nested in a distributed fashion inside a common outer loop. In our framework, each one of the multiple inner array descriptors represents the address stream for a single distributed loop, with the order of address generation proceeding from the leftmost to rightmost inner descriptor.

It is important to note that while all the descriptors in Figure 3 show two nesting levels only, our framework allows an arbitrary nesting depth. This permits describing higher-dimensional LDS traversals, for example loop nests with  $> 2$  nesting depth. Also, our framework can handle non-recurrent loads using “singleton” descriptors. For example, a pointer to a structure may be dereferenced multiple times to access different fields in the structure. Each dereference is a single non-recurrent load. We create a separate descriptor for each non-recurrent load, nest it underneath its recurrent load’s descriptor, and assign an appropriate offset value,  $O$ , and length value,  $L = 1$ .

In addition to nested composition, our framework also permits *recursive composition*. Recur-

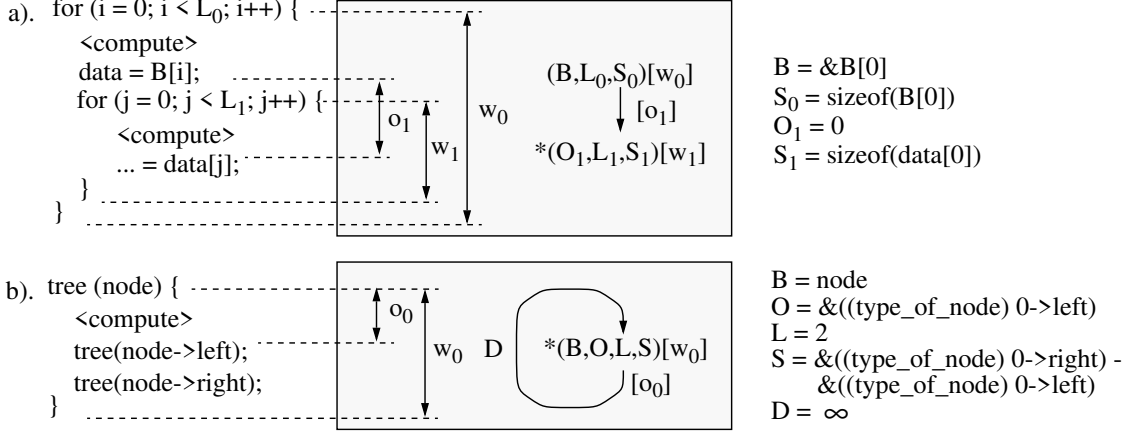


Figure 5: Each LDS descriptor graph is extended with work parameter and offset parameter annotations to specify the traversal code work information. Annotations for a). nested composition and b). recursive composition are shown. Each example shows the descriptor graph with annotations. In addition, the amount of work associated with each annotation parameter is indicated next to the source code.

sively composed descriptors describe depth-first tree traversals. They are similar to nested descriptors, except the dependence edge flows backwards. Since recursive composition introduces cycles into the descriptor graph, our framework requires each backwards dependence edge to be annotated with the depth of recursion,  $D$ , to bound the size of the data structure. Figure 4 shows a simple recursive descriptor in which the backwards dependence edge originates from and terminates to a single array descriptor. The “ $L$ ” parameter in the descriptor specifies the fanout of the tree. In our example,  $L = 3$ , so the traversed data structure is a tertiary tree, as shown in Figure 4. Notice the array descriptor has both  $B$  and  $O$  parameters— $B$  provides the base address for the first instance of the descriptor, while  $O$  provides the offset for all recursively nested instances.

In Figures 2 and 4, we assume the  $L$  parameter for linked lists and the  $D$  parameter for trees are known a priori, which is generally not true. Later in Section 4.3, we discuss how our framework handles these unknown descriptor parameters. In addition, our prefetch engine, discussed in Section 5, does not require these parameters, and instead prefetches until it encounters a null pointer.

### 3.2 Traversal Code Work Information

The traversal code work information specifies the amount of work performed by the application code as it traverses the LDS. Our LDS descriptor framework extends the descriptor graph with annotations to specify the traversal code work information. Two types of annotations are provided. First, the *work parameter annotation* specifies the work per iteration of a traversal loop in cycles. Each array or linked list descriptor in an LDS descriptor graph contains a work parameter annotation, appearing inside brackets at the end of the descriptor. Figure 5 illustrates the work parameter annotations for nested and recursively composed descriptors. In Figure 5a,  $w_0$  and  $w_1$  specify the work per iteration of the outer and inner loops, respectively, in a nested composition. For nested compositions, each work parameter includes the work of all nested loops, so  $w_0$  includes the work of an entire inner loop instance,  $L_1 * w_1$ . In Figure 5b,  $w_0$  specifies the work per instance of each recursive function call in a recursive composition. For recursive compositions, each work

parameter includes the work of a single function call instance only, and excludes all recursively called instances. Our scheduling algorithm computes the work of the recursive instances through *recursive descriptor unrolling*, discussed later in Section 4.1.2.

The second type of annotation, called the *offset parameter annotation*, specifies the work separating the first iteration of a descriptor from each iteration of its parent descriptor. Each dependence edge in an LDS descriptor graph contains an offset parameter annotation, appearing inside brackets beside the dependence edge. Figures 5a and b illustrate the offset parameter annotations,  $o_1$  and  $o_0$ , for the nested and recursively composed descriptors, respectively.

Values for both the work and offset parameters are architecture dependent, and measuring them would require detailed simulation. We make the simplifying assumption that CPI (assuming all memory references take 1 cycle as a result of perfect prefetching) is 1.0, and use static instruction counts to estimate the parameter values. While this provides only an estimate, we find the estimates are sufficiently accurate. Furthermore, in many LDS traversals, there may be multiple paths through the traversal code. When multiple paths exist, the work and offset parameters are computed assuming the shortest path, thus yielding minimum work and offset parameter values. Choosing minimum values for the traversal code work information tends to schedule prefetches earlier than necessary. While this guarantees that prefetches never arrive late, early prefetches can have negative performance consequences. Later in Section 8, we will evaluate the performance impact of early prefetches.

## 4 Prefetch Chain Scheduling

The LDS descriptor framework, presented in Section 3, captures all the information necessary for multi-chain prefetching. In the next two sections, we discuss how this information is used to perform prefetching. We begin by presenting an algorithm for computing a prefetch chain schedule from an LDS descriptor graph. Sections 4.1 and 4.2 describe our basic scheduling algorithm, and demonstrate how it works using a detailed example. Our basic scheduling algorithm computes an exact prefetch chain schedule which requires the size of all data structures to be known a priori, including the length of linked lists and the depth of trees (*i.e.* the  $L$  and  $D$  parameters from Section 3.1). In Sections 4.1 and 4.2, we optimistically assume these parameters are available to facilitate a clean exposition of our scheduling algorithm. Then, in Section 4.3, we discuss how our scheduling algorithm handles the more general case in which these parameters are not known.

We note our scheduling algorithm is quite general. While Section 2 illustrated the multi-chain prefetching idea using a simple array of lists example, the scheduling algorithm presented in this section can handle any arbitrary data structure composed of lists, trees, and arrays. However, our technique works only for static traversals. Later, in Section 8.7, we will discuss extensions for dynamic traversals.

### 4.1 Scheduling Algorithm

This section presents our basic scheduling algorithm. Section 4.1.1 describes the algorithm assuming acyclic LDS descriptor graphs. Then, Section 4.1.2 discusses how our scheduling algorithm handles LDS descriptor graphs that contain cycles.

```

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

```

Figure 6: The basic scheduling algorithm for acyclic descriptor graphs.

#### 4.1.1 Scheduling for Acyclic Descriptor Graphs

Figure 6 presents our basic scheduling algorithm that computes the scheduling information necessary for prefetching. Given an LDS descriptor graph, our scheduling algorithm computes three scheduling parameters for each descriptor  $i$  in the graph: whether the descriptor requires *asynchronous* or *synchronous* prefetching, the pre-traversal time,  $PT_i$  (*i.e.* the number of cycles in advance that the prefetch engine should begin prefetching a chain of pointers prior to the processor’s traversal), and the prefetch distance,  $PD_i$  (*i.e.* the pre-traversal time in terms of iterations of the loop containing the pointer-chain traversal). These parameters were introduced in Section 2.1. As we will explain below, the scheduling parameters at each descriptor  $i$  are dependent upon the parameter values in the sub-graph nested underneath  $i$ ; hence, descriptors must be processed 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, as is done in Figures 3 and 5 (and later in Figure 7). As mentioned earlier, our basic scheduling algorithm assumes there are no cycles in the descriptor graph. Our basic 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. First, we determine whether a descriptor requires asynchronous or synchronous prefetching. We say 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 we say it requires synchronous prefetching.<sup>1</sup> The “if” conditional test in Figure 6 computes whether asynchronous or synchronous prefetching is used.

Next, we compute the pre-traversal time,  $PT_i$ . The computation of  $PT_i$  is different for asynchronous prefetching and synchronous prefetching. 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  $PT = 3l - 2w_1$  for a 3-iteration pointer-chasing loop. If we generalize this expression, we get  $PT_i = L_i * (l - w_i) + w_i$ . In contrast, 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 6 compute  $PT_i$  for asynchronous and synchronous prefetching, respectively. Notice these equations both contain an extra term,  $PT_{nest_i}$ .  $PT_{nest_i}$  serializes

<sup>1</sup>This implies that linked list traversals in which  $l \leq w_i$  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.

$PT_i$  and  $PT_k$ , where  $PT_k$  is the pre-loop time of any descriptor  $k$  nested underneath descriptor  $i$  using indirection (*i.e.* the nested composition illustrated in Figure 3b). 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 6 considers all descriptors composed under descriptor  $i$  that use indirection and sets  $PT_{nest_i}$  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-traversal time, which is  $\lceil \frac{PT_i}{w_i} \rceil$ . Equations 3 and 5 in Figure 6 compute the prefetch distance for asynchronous and synchronous prefetching, respectively.

#### 4.1.2 Recursive Descriptor Unrolling

Cycles can occur in LDS descriptor graphs due to recursive composition, as discussed in Section 3.1. To handle cyclic descriptor graphs, we remove the cycles to obtain an equivalent acyclic descriptor graph, and apply the basic scheduling algorithm for acyclic descriptor graphs introduced in Section 4.1.1. The technique used to remove the cycles is called *recursive descriptor unrolling*.

Recursively composed descriptors can be “unrolled” by replicating all descriptors along the cycle  $D$  times, where  $D$  is the depth of recursion associated with the backwards dependence edge. Each replicated descriptor is nested underneath its parent in a recursive fashion, preserving the LDS traversal order specified by the original recursively composed descriptor.

In addition to descriptor replication, recursive descriptor unrolling must also update the work parameter annotations associated with replicated descriptors. Specifically, each work parameter along the chain of replicated descriptors should be increased to include the work of all nested descriptors created through replication. To update the work parameters, we visit each replicated descriptor in the unrolled descriptor graph in bottom-up order. For each visited descriptor, we sum the work from all child descriptors into the visited descriptor's work parameter, where each child descriptor  $i$  contains work  $L_i * w_i$ .

## 4.2 Example of an Exact Prefetch Chain Schedule

This section presents a detailed prefetch chain scheduling example using the techniques described in Section 4.1. Figure 7 illustrates each step of our technique applied to the example, and Figure 8 graphically shows the final prefetch chain schedule computed by our scheduling algorithm. In this example, we assume the size of data structures is known, allowing our scheduling algorithm to compute an exact prefetch chain schedule.

We have chosen the traversal of a tree of lists data structure as our example. This particular LDS traversal resembles the computation performed in the Health benchmark from the Olden suite [29], one of the applications used in our performance evaluation presented in Section 8. The tree of lists data structure appears in Figure 7a. We assume a balanced binary tree of depth 4 in which each tree node contains a linked list of length 2.<sup>2</sup> Furthermore, we assume a depth-first

---

<sup>2</sup>Our tree of lists data structure differs slightly from the one in the Health benchmark. Health uses a balanced quad tree of depth 5 in which each tree node contains several variable-length linked lists. One of the linked lists, which causes most of the cache misses in Health, contains roughly 150 link elements.

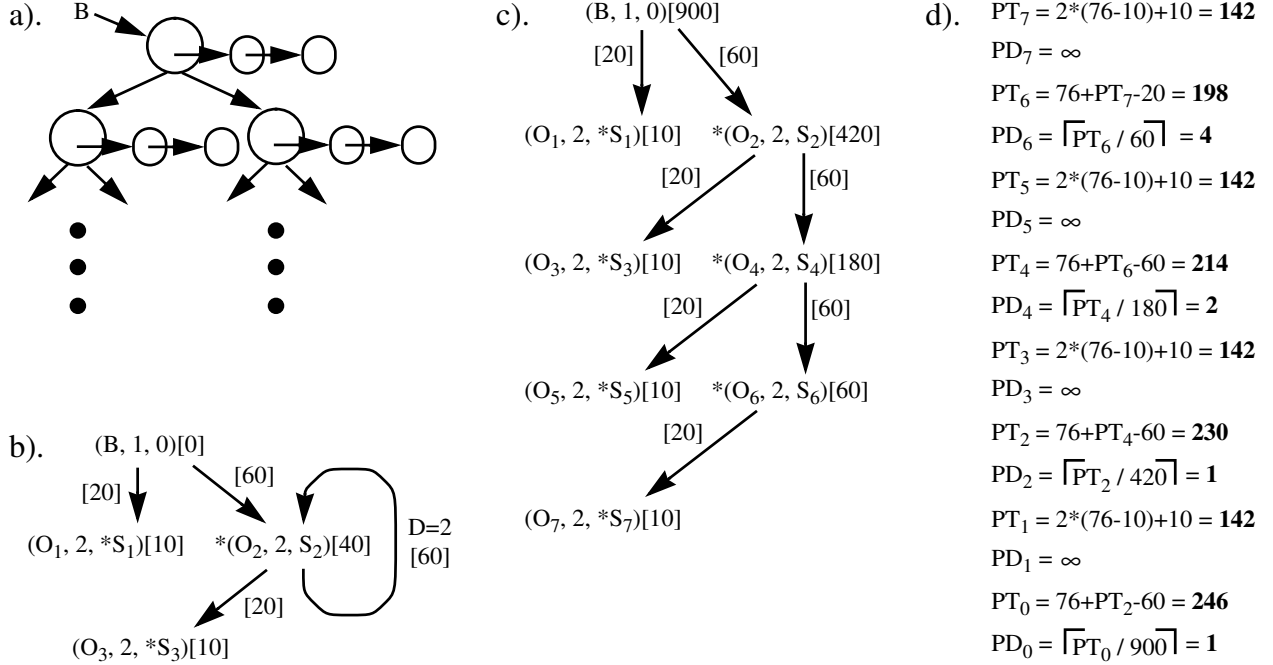


Figure 7: Prefetch chain scheduling example for a tree of lists traversal. a). Tree of lists data structure. b). Cyclic LDS descriptor graph representation. c). Equivalent acyclic LDS descriptor graph after unrolling. d). Scheduling parameter solutions using  $l = 76$  cycles.

traversal of the tree in which the linked list at each tree node is traversed before the recursive call to the left and right child nodes. (Note Figure 7a only shows two levels of our depth-four tree.)

First, we extract data layout and traversal code work information via static analysis of the application code (this is the compiler’s job, and will be described later in Section 6). Figure 7b shows the LDS descriptor graph resulting from this analysis. Descriptor 0, the root of the descriptor graph, is a “dummy” descriptor that provides the root address of the tree, “B.” Descriptors 1 and 2, nested directly underneath descriptor 0, traverse the linked list and tree node, respectively, at the root of the tree. Composed under descriptor 2 are the descriptors that traverse the linked list (descriptor 3 which is identical to descriptor 1) and tree node (descriptor 2 itself composed recursively) at the next lower level of the tree. The data layout information in this descriptor graph is exactly the information used by our prefetch engine, described later in Section 4.

Second, we apply recursive descriptor unrolling to the cyclic descriptor graph in Figure 7b to remove cycles. Figure 7c shows the equivalent acyclic descriptor graph after unrolling. Descriptor replication creates two copies of the recursively composed descriptors since  $D = 2$  in Figure 7b, producing descriptors 4 and 6 (copies of descriptor 2), and descriptors 5 and 7 (copies of descriptor 3). After copying, the work parameters are updated to reflect the work in the newly created descriptors as described in Section 4.1.2.

Finally, we compute the pre-loop time,  $PT_i$ , and the prefetch distance,  $PD_i$ , for each descriptor  $i \in \{0..7\}$  in Figure 7c using the basic scheduling algorithm. Figure 7d shows the computation of the scheduling parameters for each descriptor in bottom-up order, following the algorithm in Figure 6. For this computation, we assume an arbitrary memory latency,  $l$ , of 76 cycles.

Figure 8 graphically displays the prefetch chain schedule specified by the scheduling parameters in Figure 7d by plotting the initiation and completion of prefetch requests from different descriptors

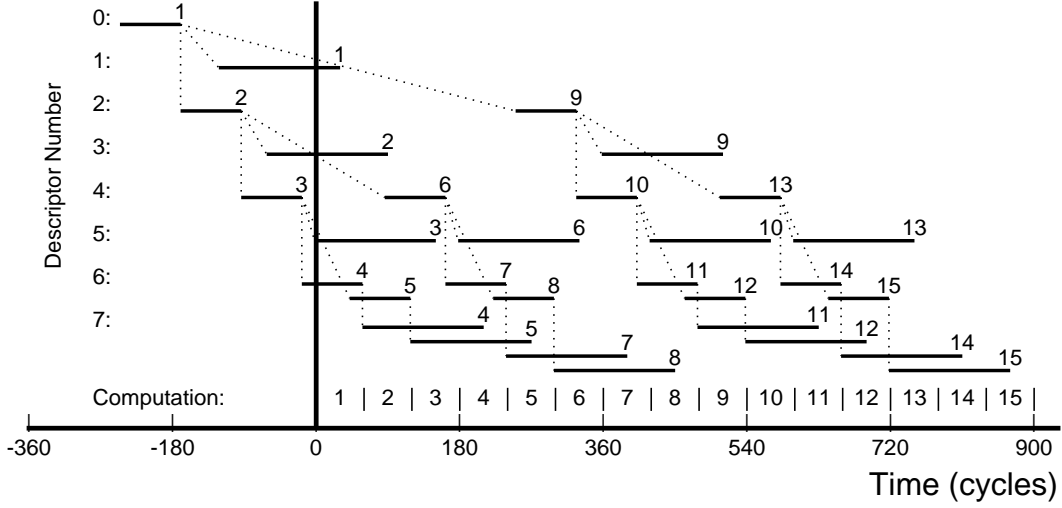


Figure 8: A graphical display of the final prefetch chain schedule computed by our scheduling algorithm for the tree of lists example. Short horizontal lines denote prefetches for the tree nodes, long horizontal lines denote prefetches for the linked lists, and dotted lines indicate address generation dependences. The “Computation” timeline shows the traversal of the tree nodes by the application code. We assume all prefetches take 76 cycles to complete.

( $y$ -axis) as a function of time ( $x$ -axis). Short solid horizontal lines denote prefetches for the tree nodes, long solid horizontal lines denote 2 serialized prefetches for each linked list, and dotted lines indicate address generation dependences. The “Computation” timeline shows the traversal of the tree nodes by the application code, with a unique ID assigned to each instance of a recursive function call. Each prefetch request has been labeled with the function call instance ID that consumes the prefetched data. Notice all prefetch requests complete by the time the processor consumes the data. After an initial startup period which ends at  $Time = 0$ , the processor will not have to stall. The figure also shows our scheduling algorithm exposes significant inter-chain memory parallelism in the tree of lists traversal.

### 4.3 Handling Unknown Descriptor Graph Parameters

Throughout Sections 4.1 and 4.2, we have assumed that LDS descriptor graph parameters are known statically, enabling our scheduling algorithm to compute an exact prefetch chain schedule off line. This assumption is optimistic since certain LDS data layout parameters may not be known until runtime. In particular, static analysis typically cannot determine the list length,  $L$ , in linked list descriptors, and the recursion depth,  $D$ , in recursively composed descriptors (see Section 3.1 for a description of these parameters). As a consequence, it is impossible to compute an exact prefetch chain schedule statically for many LDS traversals.

One potential solution is to obtain the missing information through profiling. Unfortunately, this approach has several drawbacks. First, profiling can be cumbersome because it requires profile runs, and the accuracy of the profiles may be sensitive to data inputs. Furthermore, profiling cannot determine the data layout parameters exactly if the size of data structures varies during program execution. Rather than using profiles, we propose another solution that uses only static information. The key insight behind our solution is that the scheduling parameters computed by

$$\begin{aligned}
PT_7(L) &= L(76 - 10) + 10 = 66L + 10 \\
PD_7 &= \infty \\
PT_6(L) &= 76 + PT_7 - 20 = 66L + 66 \\
PD_6(L) &= \left\lceil \frac{PT_6(L)}{10L + 40} \right\rceil = \left\lceil \frac{66L + 66}{10L + 40} \right\rceil \\
PD_6(L \rightarrow \infty) &\rightarrow \left\lceil \frac{66}{10} \right\rceil = 7
\end{aligned}$$

Figure 9: Computing scheduling parameters for the leaf nodes as a function of the list length,  $L$ , for the tree of lists example.

our scheduling algorithm are *bounded* regardless of the size of dynamic data structures. Hence, our scheduling algorithm can compute a bounded solution when an exact solution cannot be determined due to incomplete static information.

To illustrate how bounded prefetch chain schedules can be computed for linked lists and trees, consider once again the tree of lists example from Figure 7. We will first assume the length of each linked list (the  $L$  parameter in descriptors 1 and 3 in Figure 7b) is not known. Then, we will assume the depth of recursion (descriptor 2’s  $D$  parameter in Figure 7b) is not known.

#### 4.3.1 Unknown List Length

Without the list length, the pre-loop time of the linked list descriptors ( $PT_7$ ,  $PT_5$ ,  $PT_3$ , and  $PT_1$  in Figure 7d) cannot be computed, which in turn prevents computing the prefetch distance for the tree node descriptors ( $PD_6$ ,  $PD_4$ ,  $PD_2$ , and  $PD_0$ ). However, it is possible to express these quantities as a function of the list length. Figure 9 shows the computation for descriptors 6 and 7 from Figure 7d as a function of the list length,  $L$ .

In Figure 9, the prefetch distance for the list,  $PD_7$ , remains  $\infty$  because this linked list should be prefetched as fast as possible independent of its length. However, the prefetch distance for the tree node,  $PD_6$ , becomes a function of  $L$ . Notice in the limit, as  $L \rightarrow \infty$ ,  $PD_6 \rightarrow \left\lceil \frac{66}{10} \right\rceil = 7$ . Such a bounded prefetch distance exists because both the pre-loop time of the list ( $66L + 66$ ) and the work of its parent loop ( $10L + 40$ ) depend linearly on the list length. Consequently,  $PD_6$  does not grow unbounded, but instead grows towards an asymptotic value as list length increases. In general, the prefetch distances for the predecessors of a linked list descriptor in an LDS descriptor graph are always bounded, and can be determined statically by solving the scheduling equations in the limit, as illustrated in Figure 9.

Although the prefetch distances associated with linked lists are bounded, buffering requirements may not be. A bounded prefetch distance only fixes the number of lists prefetched simultaneously; the number of prefetches issued in advance of the processor for each list grows linearly with its length. Hence, buffering for prefetched data is in theory unbounded. Applications requiring large amounts of buffering may suffer thrashing, limiting performance. Fortunately, we find lists are relatively short and do not grow unbounded in practice, most likely because long lists are undesirable from an algorithmic standpoint. Programmers typically employ data structures such as hash tables and trees to reduce chain length, and thus reduce buffering requirements. In Section 8, we will evaluate the impact of finite buffering on multi-chain prefetching performance.

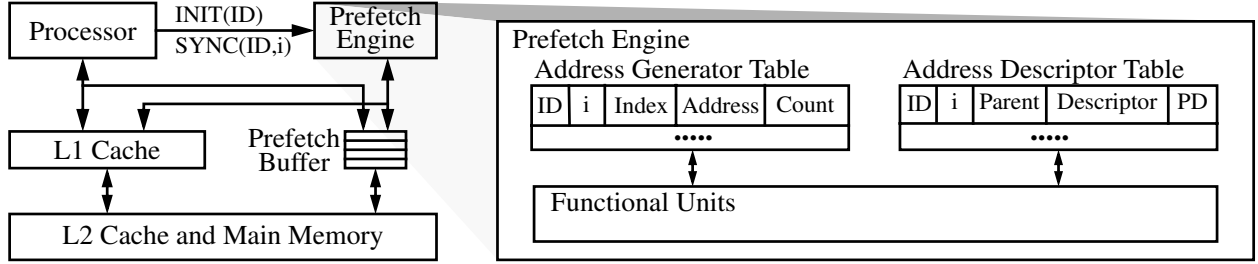


Figure 10: Prefetch engine hardware and integration with a commodity microprocessor.

### 4.3.2 Unknown Recursion Depth

Without the recursion depth, recursively composed descriptors (*e.g.* descriptors 2 and 3 in Figure 7b) cannot be unrolled, and their scheduling parameters thus cannot be computed. However, the prefetch distance of any descriptor created during unrolling is guaranteed not to exceed the prefetch distance of the leaf descriptor (*e.g.* descriptor 7 in Figure 7d). This is because the leaf descriptor contains the least amount of work among all descriptors created during unrolling; hence, it has the largest prefetch distance. So, the leaf descriptor’s prefetch distance is an upper bound on the prefetch distances for all other descriptors associated with the tree traversal. For example, in Figure 7d, the prefetch distance of the leaf descriptor,  $PD_6$ , is 4, while the prefetch distances for its predecessor descriptors,  $PD_4$ ,  $PD_2$ , and  $PD_0$  are only 2, 1, and 1, respectively.

Furthermore, the leaf descriptor’s prefetch distance is independent of recursion depth. This is because a descriptor’s predecessors in the LDS descriptor graph do not affect any of its data layout or traversal code work parameters. In particular, regardless of the recursion depth, the traversal code work associated with the leaf descriptor is exactly the amount of work in a single function call instance since there are no other recursive calls beyond a leaf node. Consequently, all LDS descriptor graph parameters necessary to compute the prefetch distance for the leaf descriptor are available statically. So, our scheduling algorithm can compute the leaf descriptor’s prefetch distance exactly, which yields the bound on the prefetch distance for all descriptors contained in the tree traversal as well, despite the fact that the recursion depth is not known.

We propose to handle unknown recursion depth by computing the prefetch distance for the leaf descriptor, and then using this prefetch distance for descriptors at all levels in the tree traversal. This approach ensures that nodes at the leaves of the tree are prefetched using an exact prefetch distance, and that all other tree nodes are prefetched using a prefetch distance that is larger than necessary. Since a large fraction of a tree’s nodes are at the leaves of the tree, our approach should provide good performance.

## 5 Prefetch Engine

In this section, we introduce a programmable prefetch engine that performs LDS traversal outside of the main CPU. Our prefetch engine uses the data layout information described in Section 3.1 and the scheduling parameters described in Section 4.1 to guide LDS traversal.

The left of half of Figure 10 shows the integration of the prefetch engine with a commodity microprocessor. The design requires three additions to the microprocessor: the prefetch engine itself, a prefetch buffer, and two new instructions called *INIT* and *SYNC*. During LDS traversal,

the prefetch engine fetches data into the prefetch buffer if it is not already in the L1 cache at the time the fetch is issued (a fetch from main memory is placed in the L2 cache on its way to the prefetch buffer). All processor load/store instructions access the L1 cache and prefetch buffer in parallel. A hit in the prefetch buffer provides any necessary data to the processor in 1 cycle, and also transfers the corresponding cache block from the prefetch buffer to the L1 cache (in the case of a store, the write is then performed in the L1 cache).

The prefetch engine, shown in the right half of Figure 10, consists of two hardware tables, the *Address Descriptor Table* and the *Address Generator Table*, and some functional units. The rest of this section discusses how the prefetch engine works. First, Sections 5.1 and 5.2 describe the Address Descriptor and Address Generator tables in detail, explaining how these tables are used to compute prefetch addresses from the LDS descriptors introduced in Section 3. This discussion uses the array of lists example from Figure 1; the code for this example, annotated with *INIT* and *SYNC* instruction macros, appears in Figure 11a. Then, Section 5.3 discusses how prefetches are scheduled according to the scheduling information computed by our scheduling algorithm. Finally, Section 5.4 presents a detailed implementation of the prefetch engine, including a description of the functional units, and analyzes its cost.

## 5.1 Address Descriptor Table

The Address Descriptor Table (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 4.1.1. The *Parent* field specifies the descriptor’s parent in the descriptor graph. The *Descriptor* field stores all the parameters associated with the descriptor such as the base, length, and stride, and whether or not indirection is used. Finally, the *PD* field stores the prefetch distance computed by our scheduling algorithm for descriptor *i*. Figure 11b shows the contents of the ADT for our array of lists example, where *ID* = 4.

Before prefetching can commence, the ADT must be initialized with the data layout and prefetch distance information for the application. We memory map the ADT and initialize its contents via normal store instructions. Most ADT entries are filled at program initialization time. However, some ADT parameters are unknown until runtime (*e.g.* the base address of a dynamically allocated array). Such runtime parameters are written into the ADT immediately prior to each *INIT* instruction. Although Figure 11a does not show the ADT initialization code, our compiler described later in Section 6 generates all the necessary code to initialize the ADT.

Notice the ADT contents should be saved and restored across process switches by the operating system. On a process switch, the OS must flush the AGT to terminate prefetching (the next section explains the AGT), and save out the contents of the ADT. When the process is rescheduled, its ADT contents will be restored, but we assume the AGT contents will not. This simplifies the save and restore operation, but it can prematurely terminate prefetching. Since process switches are infrequent, these effects should not impact performance significantly (for frequent exceptions like TLB faults that do not result in a process switch, the OS does not have to save and restore the ADT). In our benchmarks, system calls are extremely infrequent; moreover, we only consider single-program workloads. Consequently, our simulator assumes both the ADT and AGT contents persist across all context switches and does not model switching-related overheads.

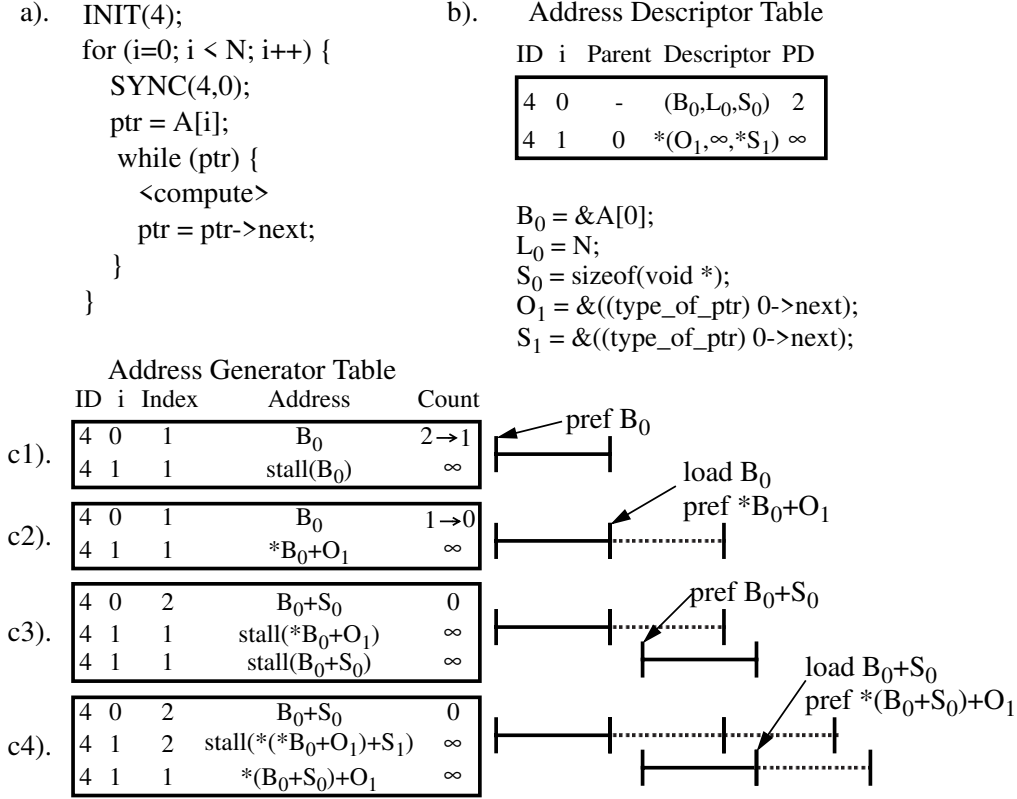


Figure 11: LDS traversal example. a). Array of lists traversal code annotated with prefetch directives. b). ADT contents. c). AGT contents at 4 different times during LDS traversal.

## 5.2 Address Generator Table

The Address Generator Table (AGT) generates the LDS traversal address stream specified by the data layout information stored in the ADT. AGT entries are activated dynamically. Once activated, each AGT entry generates the address stream for a single LDS descriptor. AGT entry activation can occur in one of two ways. First, the processor can execute an *INIT*(*ID*) instruction to initiate prefetching for the data structure identified by *ID*. Figure 11c1 shows how executing *INIT*(4) activates the first entry in the AGT. The prefetch engine searches the ADT for the entry matching *ID* = 4 and *i* = 0 (*i.e.* entry (4,0) from Figure 11b) which is the root node for descriptor graph #4). An AGT entry (4,0) is allocated for this descriptor, the *Index* field is set to one, and the *Address* field is set to *B*<sub>0</sub>, the base parameter from ADT entry (4,0). Once activated, AGT entry (4,0) issues a prefetch for the first array element at address *B*<sub>0</sub>, denoted by a solid bar in Figure 11c1.

Second, when an active AGT entry computes a new address, a new AGT entry is activated for every node in the descriptor graph that is a child of the active AGT entry. (Note, address computation for the AGT occurs in the functional units shown in Figure 10; these will be described in detail in Section 5.4). As shown in Figure 11c1, a second AGT entry, (4,1), is activated after AGT entry (4,0) issues its prefetch because (4,0) is the parent of (4,1) in the ADT. This new AGT entry is responsible for prefetching the first linked list; however, it stalls initially because it must wait for the prefetch of *B*<sub>0</sub> to complete before it can compute its base address, \**B*<sub>0</sub>. Eventually, the prefetch of *B*<sub>0</sub> completes, AGT entry (4,1) loads the value, and issues a prefetch for address \**B*<sub>0</sub>, denoted by a dashed bar in Figure 11c2.

Figures 11c3 and 11c4 show the progression of the array of lists traversal. In Figure 11c3, AGT entry (4,0) generates the address and issues the prefetch for the second array element at  $B_0 + S_0$ . As a result, its *Index* value is incremented, and another AGT entry (4,1) is activated to prefetch the second linked list. Once again, this entry stalls initially, but continues when the prefetch of  $B_0 + S_0$  completes, as shown in Figure 11c4. Furthermore, Figures 11c3 and 11c4 show the progress of the original AGT entry (4,1) as it traverses the first linked list serially. In Figure 11c3, the AGT entry is stalled on the prefetch of the first link node. Eventually, this prefetch completes and the AGT entry issues the prefetch for the second link node at address  $*B_0 + S_1$ . In Figure 11c4, the AGT entry is waiting for the prefetch of the second link node to complete.

AGT entries are deactivated once the *Index* field in the AGT entry reaches the *L* parameter in the corresponding ADT entry, or in the case of a pointer-chasing AGT entry, if a null pointer is reached during traversal.

### 5.3 Prefetch Scheduling

When an active AGT entry generates a new memory address, the prefetch engine must schedule a prefetch for the memory address. Prefetch scheduling occurs in two ways. First, if the prefetches for the descriptor should issue asynchronously (*i.e.*  $PD_i = \infty$ ), the prefetch engine issues a prefetch for the AGT entry as long as the entry is not stalled. 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. The (4,1) AGT entries in Figure 11 are scheduled in this fashion.

Second, if the prefetches for the descriptor should issue synchronously (*i.e.*  $PD_i \neq \infty$ ), then the prefetch engine synchronizes the prefetches with the code that traverses the corresponding array or linked list. We rely on the compiler 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, as shown in Figure 11a. 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, and is decremented each time the prefetch engine issues a prefetch for the AGT entry, as shown in Figures 11c1 and 11c2. In addition, the prefetch engine “listens” for *SYNC* instructions. When a *SYNC* executes, it emits a descriptor graph *ID* and a descriptor number *i* that matches an AGT entry (essentially matching up the loop with the AGT entry generating its address stream). 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, as it has in Figure 11c2, 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 (not shown in Figure 11).

### 5.4 Hardware Implementation and Cost

Figure 12 presents an implementation of the prefetch engine. This figure is similar to Figure 10, but illustrates the ADT, AGT, and functional units in greater detail. Enough detail has been provided to permit an estimate of the prefetch engine’s hardware cost.

In Figure 12, the ADT and AGT are identical to the ones in Figure 10 except for two differences. First, the AGT *Descriptor* field has been expanded to show the descriptor parameters. The *B*, *L*, *S*, and *O* parameters are the same as those described in Section 3.1; the “\*S” parameter

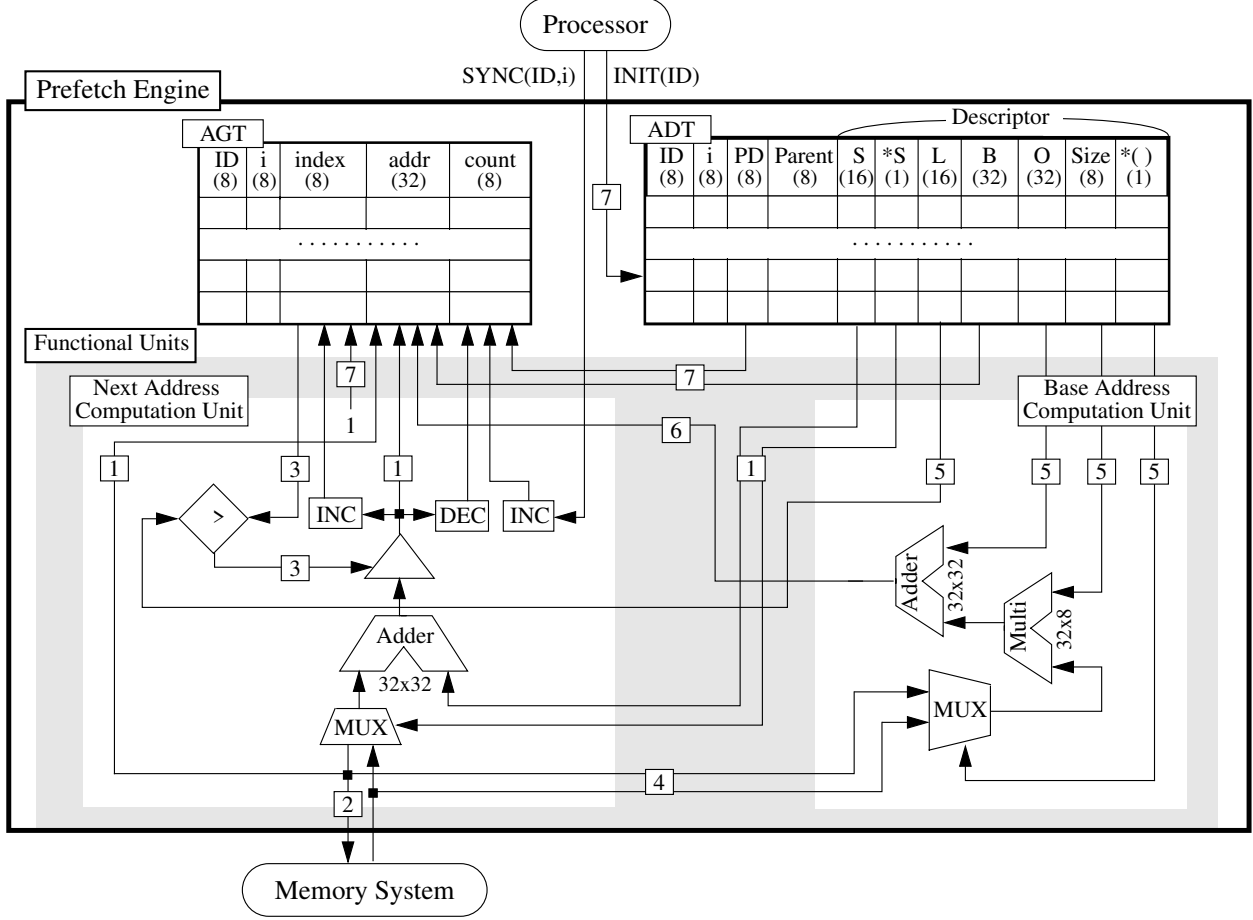


Figure 12: Prefetch engine implementation, detailing the ADT and AGT, and two functional units, the Next Address Computation Unit and the Base Address Computation Unit.

specifies whether the descriptor is an array descriptor or a linked-list descriptor; the “\*( )” parameter indicates whether indirection is used to generate the base address of a nested descriptor; and the “Size” parameter is used during base address computation for nested descriptors (this parameter is needed for indexed array addressing only). Second, the size (in bits) of all ADT and AGT fields have been specified in parentheses.

From the field sizes in Figure 12, we see that each ADT entry is 18 bytes wide, while each AGT entry is 8 bytes wide. To determine the total size of the ADT and AGT, we must also select the number of entries. The number of ADT entries should be selected to accommodate the LDS descriptors, and the number of AGT entries should be selected to accommodate AGT activation. We have found that for the benchmarks used in our evaluation, 75 and 128 entries are sufficient for the ADT and AGT, respectively. Consequently, the ADT is 1.32 Kbytes and the AGT is 1 Kbytes. (Later, in Section 8.5, we will vary table size and study the impact on performance).

Figure 12 also shows two functional units, the *Next Address Computation Unit* (NACU) and the *Base Address Computation Unit* (BACU), for computing addresses during LDS traversal. The NACU consists of a 32x32 adder, 2 incrementers, and 1 decrements, and is responsible for computing the address stream for active AGT entries. Every cycle, the NACU selects a single active AGT entry, computes its next address, and writes the new address back into the AGT (wires labeled

“1”). Computed addresses are also issued to the memory system for prefetching (wire labeled “2”). In addition, the NACU compares the AGT entry’s *Index* to its corresponding *Length* parameter (wires labeled “3”), and deactivates the AGT entry if the address stream is exhausted. Every time the NACU computes a new address for an AGT entry that has nested child descriptors, it forwards the address to the BACU (wires labeled “4”). The BACU consists of a 32x8 multiplier and a 32x32 adder, and is responsible for activating AGT entries for nested descriptors (see Section 5.2 for a discussion on AGT entry activation). For each address forwarded from the NACU, the BACU computes a new base address using descriptor parameters (wires labeled “5”), and activates a new AGT entry (wire labeled “6”). As described in Section 5.2, AGT entry activation can also occur when the processor executes an *INIT(ID)* instruction; the wires labeled “7” perform this action.

Both the NACU and BACU have modest hardware complexity. An important question, however, is how many units are necessary to sustain an adequate prefetch throughput? Surprisingly, even with an 128-entry AGT, we have found that a single NACU and a single BACU is sufficient. Address computation is not the bottleneck in multi-chain prefetching. In fact, we have observed in our experiments that the prefetch engine is idle most of the time waiting for long-latency memory operations to complete. Given that address generation concurrency is not important, the functional units contribute very little to the cost of the prefetch engine.

Finally, the last major hardware component in the prefetch engine is the prefetch buffer, shown in Figure 10. The prefetch buffer should be large enough to hold all prefetched data prior to their access by the processor. Too small a prefetch buffer can lead to thrashing, defeating the benefits of prefetching. We have found that for the benchmarks used in our evaluation, a 1 Kbyte prefetch buffer provides good performance. (Later, in Sections 8.3 and 8.5, we study the thrashing problem and the performance impact of varying prefetch buffer size).

In summary, our prefetch engine consumes roughly 4.32 Kbytes of on-chip RAM. In addition, it requires combinational logic to implement the functional units, consisting of 2 32x32 adders, a 32x8 multiplier, 2 incrementers, and a decrements. Overall, we find the hardware complexity of the prefetch engine to be modest.

## 6 Compiler Support

Sections 3 through 5 presented the algorithms and hardware for performing multi-chain prefetching based on our LDS descriptor framework. An important question is how are the LDS descriptors constructed? Also, how is the software support (*i.e.* the *INIT* and *SYNC* instructions, and the ADT initialization code) instrumented? One approach is to rely on the programmer to carry out these tasks. Unfortunately, this approach is ad hoc and error prone, and requires significant programmer effort. A more desirable approach is to develop systematic algorithms for performing these tasks and to implement them in a compiler, thus automating multi-chain prefetching.

This section addresses the compiler support necessary to automate multi-chain prefetching. Our approach consists of three major steps. First, we analyze the program structure to determine *where* to initiate prefetching. Second, for each selected prefetch initiation site, we extract an LDS descriptor graph. Finally, from the extracted LDS descriptors, we compute the scheduling information following the algorithms already presented in Section 4, and instrument the application code. The following sections discuss these compiler steps in detail. Sections 6.1 and 6.2 introduce the algorithms for performing the first two steps. Then, Section 6.3 describes a prototype compiler that implements all of our algorithms, and performs the necessary software instrumentation.

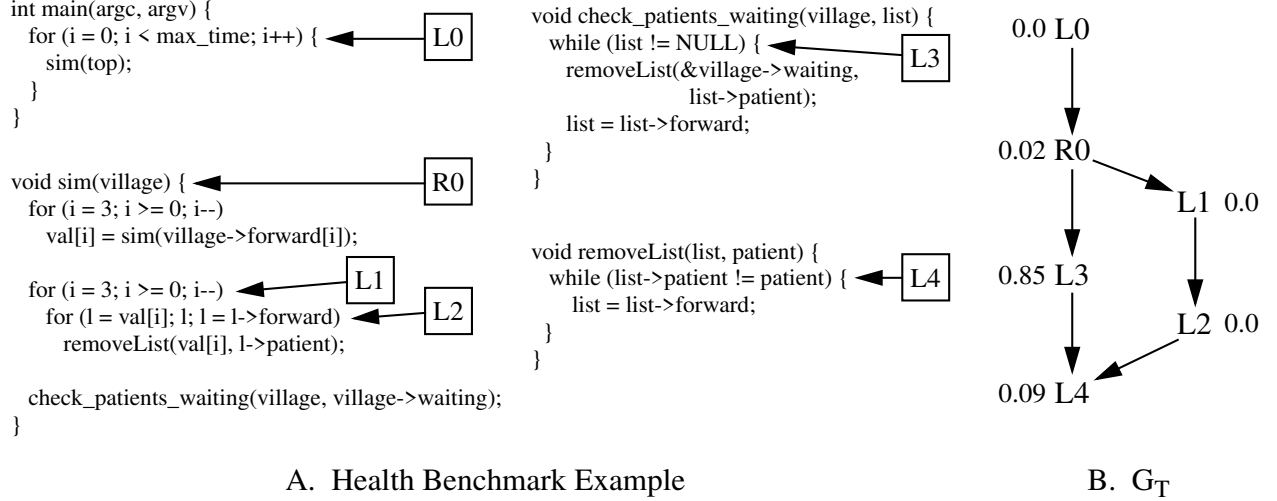


Figure 13: Constructing a traversal nesting graph. A. Kernel code from the Health benchmark. Boxed labels indicate the possible prefetch initiation sites. B.  $G_T$  annotated with cache-miss breakdowns acquired via profiling.

## 6.1 Prefetch Initiation

The first step that a compiler for multi-chain prefetching must perform is to determine where to initiate prefetching. For simplicity, we only permit prefetch initiation immediately prior to a loop or a call to a recursive function, *i.e.* at a point in the program where a new LDS traversal begins. To provide the information necessary for identifying such prefetch initiation sites, we build a data structure called the *traversal nesting graph*, or  $G_T$ .  $G_T$  is a directed acyclic graph whose nodes represent loops or recursive functions defined in the source code, and whose edges connect those nodes corresponding to loops or recursive functions that are nested. Hence,  $G_T$  reflects the nesting structure of all possible prefetch initiation sites in a program. Moreover, we construct  $G_T$  globally, so its edges capture nesting relationships across procedures as well as within procedures.

Figure 13 illustrates  $G_T$  for the Health benchmark. In Figure 13A, a kernel code from Health consisting of 4 functions is shown. For readability, only function headers, loops, and function call sites appear in the figure. The boxed labels indicate the 5 loops (“L0” through “L4”) and recursive function (“R0”) that are candidates for prefetch initiation. In Figure 13B, the corresponding  $G_T$  for this code is shown. The graph reports the nesting structure of the possible prefetch initiation sites identified in Figure 13A. (Note, in our example, we construct  $G_T$  for the kernel code only; the actual analysis would consider the entire Health benchmark).

After constructing  $G_T$ , we compute the set of prefetch initiation sites, which we call  $P$ . Figure 14 presents our algorithm for computing  $P$  given  $G_T$ . The main procedure in our algorithm, **select\_prefetch\_sites**, visits every node,  $N$ , in  $G_T$  (lines 2 and 18), and considers nodes for initiating prefetching based on two criteria. First, only nodes corresponding to LDS traversals that incur a significant number of cache misses are considered. This minimizes runtime overhead, thus increasing the potential performance gain of prefetching. Unfortunately, accurately predicting cache-miss behavior from static information alone (*e.g.*  $G_T$ ) is difficult; hence, we augment our static analysis with profiling. We built a profiling tool, which we will describe in Section 6.3, that breaks down the total cache misses into the fraction incurred by each loop and recursive function, and annotates the corresponding nodes in  $G_T$  with the breakdown values. As an example, the

<b>select_prefetch_sites(<math>G_T</math>)</b> Given: traversal nesting graph, $G_T$ Compute: prefetch initiation sites, $P$	<b>no_gain(<math>N</math>)</b> Given: $G_T$ node, $N$ Compute: potential for gain	<b>overlap(<math>N, P</math>)</b> Given: $G_T$ node, $N$ , set of prefetch initiation sites, $P$ Compute: nesting conflict
<hr/> 1: $P = \Phi$ ; 2: do { 3:   if (no_gain( $N$ )    too_complex( $N$ )) 4:     continue; 5:   if (recursive( $N$ ) && (fanout( $N$ ) > 1)) { 6:     if (!(overlap( $N, P$ ))) 7: $P = P \cup \{N\}$ ; 8:   } else { 9: $Q = \Phi$ ; 10:    do { 11:     if (!(too_complex( $M$ )) && !(overlap( $M, P \cup Q$ ))) 12: $Q = Q \cup \{M\}$ ; 13:    } $\forall M$ that is a parent of $N$ in $G_T$ ; 14:    if (( $Q == \Phi$ ) && !(overlap( $N, P$ ))) 15: $Q = \{N\}$ ; 16: $P = P \cup Q$ ; 17:    } 18: } $\forall N$ in $G_T$ from most to least cache_miss( $N$ );	<hr/> 19: if ((cache_miss( $N$ ) < 0.01)    20:   (no_pointer_references( $N$ ))) 21:   return TRUE; 22: else 23:   return FALSE;  <b>too_complex(<math>N</math>)</b> Given: $G_T$ node, $N$ Compute: too complex for prefetch engine? <hr/> 24: if (no_loop_induction( $N$ )    25:   contains_goto_statements( $N$ )    26:   complex_address_generation( $N$ )    27:   data_dependent_traversal( $N$ )) 28:   return TRUE; 29: else 30:   return FALSE;	<hr/> 31: do { 32:   if (nested( $N, S$ )    nested( $S, N$ )) 33:     return TRUE; 34: } $\forall S$ in $P$ ; 35: return FALSE;

Figure 14: Algorithm for computing the set of prefetch initiation sites,  $P$ , from the traversal nesting graph,  $G_T$ .

cache-miss breakdowns acquired by our profiling tool for Health are annotated in the  $G_T$  from Figure 13B. For every node in  $G_T$  visited by `select_prefetch_sites`, our algorithm examines the corresponding cache-miss breakdown annotation (line 19). If this value is below 1%, we skip over the node and do not consider it further for prefetch initiation (lines 3 and 4).<sup>3</sup>

Second, only nodes corresponding to LDS traversals that are implementable in the prefetch engine are considered for prefetch initiation. As described in Section 5, our prefetch engine can only perform static traversals involving simple address computation and memory indirection. Hence, our algorithm analyzes the loops and recursive functions corresponding to each visited node in  $G_T$ , and skips over nodes that perform traversals too complex for our prefetch engine (lines 3 and 4). Specifically, we skip nodes that perform unstructured traversals (lines 24 and 25), traversals containing complex address computations (line 26), and data dependent traversals (line 27).

For every node,  $N$ , that satisfies both our performance and complexity criteria, we choose one or more nodes in  $G_T$  to initiate prefetching for  $N$ . The primary factor affecting this choice is memory parallelism: prefetching should be initiated from a point where sufficient memory parallelism exists to tolerate the (potentially serialized) cache misses incurred within  $N$ . The node that best provides this memory parallelism depends on  $N$ ’s LDS traversal type. If node  $N$  represents a recursive function that pursues multiple child pointers at each level of recursion (*i.e.* its “fanout” is greater than 1), then we assume  $N$  performs a tree traversal. Tree traversals contain significant memory parallelism since the sub-trees at each tree node can be pursued in parallel. Hence, in this case, we select node  $N$  itself for initiating prefetching (lines 5 and 7). (Note, recursive functions with a fanout equal to 1 are treated as linked list traversals, and are discussed next).

The situation is different if node  $N$  represents a loop. For pointer-chasing loops that traverse linked lists (*i.e.* the induction variable update is of the form `ptr=ptr->next`), all pointer accesses in  $N$  are serialized, and there is no memory parallelism. For affine loops that traverse non-recursive ribs (*i.e.* the induction variable update is of the form `i+=constant`, but the loop body dereferences

<sup>3</sup>We chose 1% arbitrarily; any small fraction would suffice. Our intention is simply to skip over traversals that contribute an insignificant number of cache misses.

pointers derived from the induction variable), all pointer accesses within a single iteration of  $N$  are serialized, and there is only limited memory parallelism.<sup>4</sup> In both cases, our algorithm assumes there is insufficient memory parallelism within node  $N$  alone to effectively tolerate the pointer-chasing cache misses. Rather than select node  $N$ , our algorithm instead considers all nodes  $M$  that are parents of  $N$  in  $G_T$  (lines 10 and 13). We select every node  $M$  that meets our traversal complexity criterion (line 11), adding it to the set of prefetch initiation sites (lines 12 and 16). By initiating prefetching from  $N$ 's parents, the degree of memory parallelism is increased since *multiple* instances of  $N$  can be prefetched simultaneously. In the event that none of  $N$ 's parents meet the traversal complexity criterion, we just choose  $N$  itself (lines 14 and 15). This ensures that  $N$  will be prefetched, even if it is from a point in  $G_T$  that provides only limited memory parallelism. Lastly, in addition to the two looping cases described above, there is a third case: node  $N$  may represent a loop that contains no pointer-chasing references whatsoever (line 20). Since our technique focuses on LDS traversals, we simply skip over all such nodes (lines 3 and 4).

Finally, it is possible for our algorithm to select two or more prefetch initiation sites that are nested in  $G_T$ . Nested prefetch initiation leads to multiple traversals of the same data structures. To avoid such redundant prefetching, our algorithm calls the procedure `overlap` each time it discovers a new node for initiating prefetching (lines 6, 11, and 14). `overlap` checks if the candidate node,  $N$ , has a nesting conflict with any node currently in  $P$  (lines 31–34), and includes  $N$  in  $P$  only if no nesting conflicts are found. Note, if  $N$  is included in  $P$ , all parents of  $N$  in  $G_T$  become un prefetchable since they will have nesting conflicts with  $N$  and cannot be selected for prefetch initiation. (While the children of  $N$  similarly cannot be selected for prefetch initiation, they will get prefetched from the prefetch initiation site at  $N$ , as we will see in Section 6.2). However, since our algorithm considers nodes in descending order of their cache-miss annotations (line 18), the most important nodes are highly likely to be prefetched.

Applying our algorithm to Figure 13B, nodes “L0,” “L1,” and “L2” are skipped since their cache-miss breakdown annotations are less than 1%. The remaining nodes are considered in the order of their annotation values, “L3,” “L4,” and “R0,” to select the prefetch initiation nodes. (Note, all nodes in Figure 13B meet the complexity criteria, but ascertaining this requires analyzing more code than has been provided in Figure 13A). For “L3,” our algorithm selects “R0.” For “L4,” our algorithm tries to select “L3,” but fails because of a nesting conflict with the already selected “R0.” And for “R0,” our algorithm selects “R0” again. Hence, the final solution is  $P = \{R0\}$ .

## 6.2 LDS Descriptor Graphs

After computing the set of prefetch initiation sites,  $P$ , the next step that a compiler for multi-chain prefetching must perform is to extract the LDS descriptor graphs—one for each node,  $N$ , in  $P$ . We begin this step by constructing code fragments from which our compiler will extract the LDS descriptor graphs. Each code fragment contains the code corresponding to a node,  $N$ , in  $P$  as well as the nodes nested underneath  $N$  in  $G_T$  whose cache-miss breakdown annotations meet the minimum threshold (1%). By including all such LDS traversal codes, we ensure the LDS descriptor graph eventually extracted from the code fragment will provide prefetching for all the loops and recursive functions associated with node  $N$  that incur a large number of cache misses. It is important to note these code fragments are not part of the application; they are constructed for the sole purpose of extracting LDS descriptor graphs, and are discarded afterwards.

---

<sup>4</sup>In this case, there is memory parallelism between iterations of  $N$  since the induction variable updates are not serialized. However, if the loop executes a small number of iterations, which is common in affine loops from non-numeric programs, there is typically not enough memory parallelism for our technique to provide performance gains.

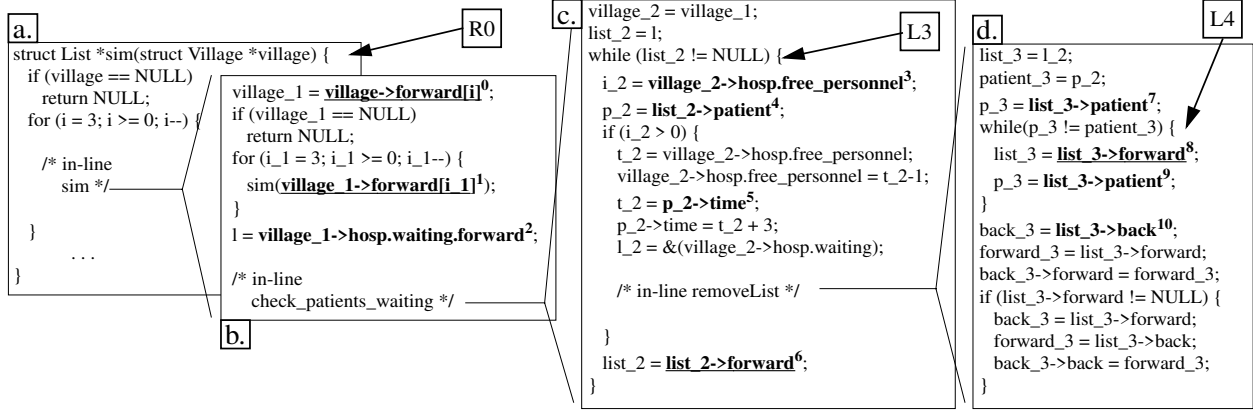


Figure 15: Construction of a code fragment for the Health benchmark. a. Code for `sim` containing “R0.” b. In-lining of recursive call to `sim`. c. In-lining of `check_patients_waiting` containing “L3.” d. In-lining of `removeList` containing “L4.”

Figure 15 illustrates the construction of a code fragment for the Health benchmark example. As described in Section 6.1, “R0” is selected as the prefetch initiation site for Health. And as shown by the  $G_T$  graph in Figure 13B, “L3” and “L4” are nested underneath “R0,” and have  $> 1\%$  cache-miss breakdown annotations. Hence, we construct a code fragment starting at `sim`, the procedure containing “R0,” and include `check_patients_waiting` and `removeList`, the procedures containing “L3” and “L4,” respectively. Figures 15a, c, and d show the contents of these procedures (they are simply more complete versions of the procedures shown in Figure 13).

As we construct the code fragment in Figure 15, we perform two types of procedure in-lining. First, we in-line calls to non-recursive procedures (e.g. `check_patients_waiting` and `removeList`) so that all the code resides in a single procedure when we’re done. This simplifies the extraction of the LDS descriptor graph later on since it relieves our compiler from having to perform inter-procedure analysis. Second, we also in-line calls to recursive procedures (e.g. `sim`) *once*. This “unrolls” two invocations of the recursive traversal, i.e. the code between the recursive procedure entry point and the recursive procedure call site(s). Our compiler analyzes the code along this traversal path to extract recursively composed descriptors. For both types of in-lining, we rename local variables to remove naming conflicts, and explicitly set any renamed caller parameters to their matching callee parameters at the top of each in-lined procedure.

Once the code fragment for a prefetch initiation site has been constructed, our compiler extracts its LDS descriptor graph. Recall from Section 3 that we must extract both data structure layout information as well as traversal code work information. In the remainder of this section, we describe the former (which is the more challenging of the two). We will discuss the latter in Section 6.3.

Extraction of the data structure layout information proceeds in 4 parts. First, we extract a *local traversal graph* for each loop or recursive function in the code fragment separately. Each local traversal graph contains a node for every memory read reference found in the corresponding loop or recursive function. Our analysis creates nodes for reads through pointers, as well as reads through arrays used to compute addresses for pointer references (we exclude array reads that do not affect pointer references since we focus on LDS traversals only). Each local traversal graph also contains a directed edge connecting every pair of nodes whose corresponding memory references have address generation dependences. For such *intra-traversal edges*, we extract the address generation functions that relate each pair of memory references connected by the edges.

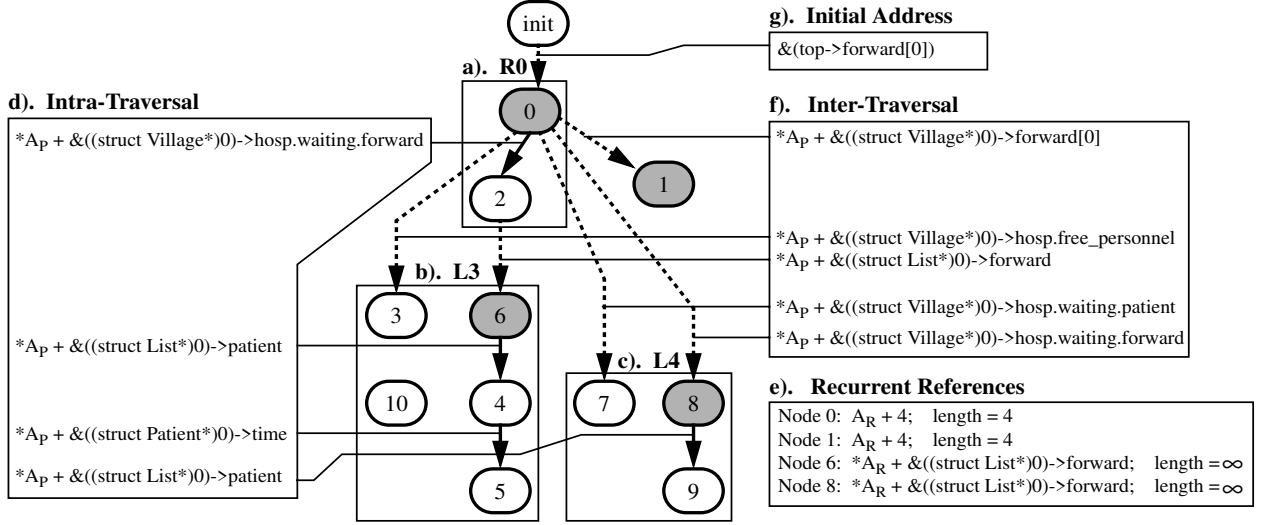


Figure 16: Data structure layout information for the Health benchmark. a-c). Local traversal graphs for “R0,” “L3,” and “L4.” d-g). Address generation functions for intra-traversal edges, recurrent references, inter-traversal edges, and initial address edges.

Figures 15 and 16 illustrate the extraction of local traversal graphs for the Health benchmark. In Figure 15, we bold-face and label the memory read references that have been identified for each loop or recursive function: “R0” contains references labeled 0 – 2, “L3” contains references labeled 3 – 6 and 10, and “L4” contains references labeled 7 – 9. In Figures 16a, b, and c, we show the local traversal graphs for “R0,” “L3,” and “L4,” respectively. The same numbers used to label the memory references in Figure 15 are used to label the corresponding graph nodes, and the solid directed edges connect the nodes corresponding to memory references with address generation dependences. In Figure 16d, we show the address generation function computed by our analysis for each directed edge. The variable “ $A_P$ ” in these functions denotes the address generated by the parent reference used in the address computation of the child reference.

Second, we analyze the code fragments for *recurrent references*. Recurrent references are memory references inside loops whose address computations depend directly on the loop induction variable. Our analysis identifies recurrent references based on loop type. In pointer-chasing loops, recurrent references are inside the expressions that update the pointer induction variable (*e.g.* `ptr=ptr->next`). In affine loops, recurrent references are the array references that use the loop induction variable as part of their array index. For each recurrent reference identified, we determine the address generation function that relates consecutive dynamic instances of the same reference. We also extract the recurrence length, or the number of iterations in the loop that contains the reference. (If the loop length is unknown, we use a default value of  $\infty$ ). In Figure 15, the recurrent references identified by our analysis are underlined, and the corresponding graph nodes in Figure 16 are shaded grey. In Figure 16e, we show the extracted address generation functions and recurrence lengths. The variable “ $A_R$ ” in the functions denotes the address generated by a single dynamic instance of the recurrent reference used in the address computation of the next dynamic instance.

Third, we insert directed edges to connect nodes from *different* local traversal graphs, corresponding to memory references with address generation dependences that span traversals. Such *inter-traversal edges* provide the “live-in values” for address generation within each local traversal graph. (Examples of such live-ins include the initial address for the first dynamic instance of a

recurrent reference, or the address for a memory reference that is invariant with respect to the local loop or recursive function). This step is analogous to inserting intra-traversal edges explained above, except instead of considering nodes within a single local traversal graph, it considers nodes from different graphs. Notice, however, for the outer-most traversal in the code fragment, there are no parent nodes from which to receive live-in values. We add a dummy `init` node to provide the initial address for such root traversals. In Figure 16, we indicate the inter-traversal edges for the Health example using dotted directed edges. Figure 16f shows the address generation functions extracted by our analysis to compute the live-in values, and Figure 16g shows the initial address provided by the `init` node.

Finally, we prune the local traversal graphs to remove unnecessary nodes. Three conditions guide pruning. First, we remove nodes without incoming edges. These nodes represent memory references whose addresses are invariant with respect to the entire code fragment; hence, prefetching them will not provide noticeable gains. Second, we perform locality analysis and remove nodes that reference an L1 cache block already referenced by another node. We also remove any edges associated with such redundant nodes. Lastly, we remove nodes created by in-lining recursive function calls. We retain each incoming edge to such nodes (as well as their address generation functions), but redirect the edge to point back at the removed node’s parent. This creates a cycle, thus forming the desired recursive composition. In Figure 16, node 10 would be removed due to the first condition. Also, node 1 would be removed due to the third condition, and its incoming edge would be redirected to point at node 0. None of the nodes in Figure 16 would be removed due to the second condition.

From Figure 16 (after pruning has been applied), a simple transformation yields the final data layout information. Replace every graph node with the descriptor  $(O, L, S)$ . Examine the address generation function associated with the node’s incoming edge. If the “ $A_P$ ” variable is preceeded by a “\*”, place a “\*” in front of the descriptor (*e.g.*  $*(O, L, S)$ ). Set  $O$  equal to this address generation function (without the “ $A_P$ ” variable). Next, if the node is not recurrent, set  $L = 1$  and  $S = 0$ . If the node is recurrent, examine the corresponding address generation function in Figure 16e. If the “ $A_R$ ” variable is preceeded by a “\*”, place a “\*” in front of the  $S$  parameter (*e.g.*  $(O, L, *S)$ ). Set  $S$  equal to this address generation function (without the “ $A_R$ ” variable) and  $L$  equal to the recurrence length. Lastly, for the node that is the child of the dummy `init` node, replace the  $O$  parameter with a  $B$  parameter (if the node is recursively composed, add the  $B$  parameter rather than replacing the  $O$  parameter). Set  $B$  equal to the initial address.

### 6.3 Prototype Compiler

We built a prototype compiler for multi-chain prefetching. Our compiler implements all the algorithms presented in Sections 6.1 and 6.2, as well as the scheduling analysis from Section 4, and inserts prefetch instrumentation into the program source code (*i.e.* the final product is a C program). Figure 17 illustrates the major modules that comprise our prototype compiler. Its construction leverages several tools from existing toolsets, including SimpleScalar [1], Unravel [22], the Stanford University Intermediate Format (SUIF) [11], and Perl. It also contains several custom-built modules implemented in C. All the steps illustrated in Figure 17 are performed automatically; no manual intervention is necessary.

Three of the modules in Figure 17 are responsible for constructing the traversal nesting graph,  $G_T$ , described in Section 6.1. The “Initiation Selector” is a binary analyzer implemented in C that extracts loops and recursive functions from the original program binary, and determines their

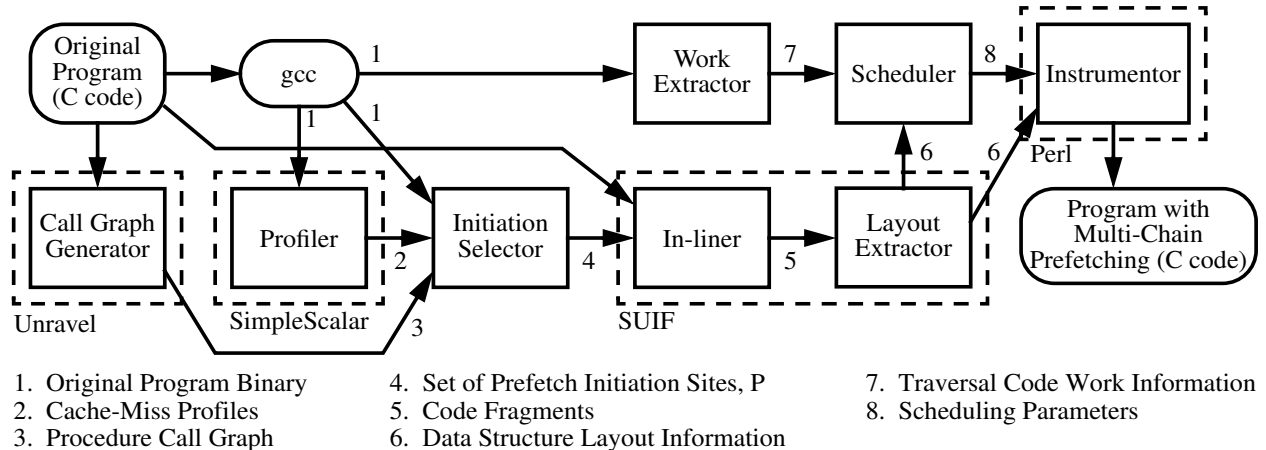


Figure 17: Modules comprising the prototype compiler for multi-chain prefetching. Dashed boxes enclose modules derived from the SimpleScalar, Unravel, SUIF, and Perl toolsets. All other modules (except for `gcc`) are custom built in C.

nesting relationships. This tool takes as input the program’s procedure call graph, which we construct using a procedure call graph generator provided as part of the Unravel toolset [22]. Based on the program’s binary and procedure call graph, the “Initiation Selector” extracts the  $G_T$  graph across the entire program. To annotate the  $G_T$  graph with the cache-miss breakdown values discussed in Section 6.1, we use cache simulation. The “Profiler” is a modified cache simulator from the SimpleScalar toolset [1] that acquires cache-miss counts on a per load PC basis. Using line number information generated by our C compiler, we map load PCs back to source code lines, and aggregate all the cache misses incurred within the same loops or recursive functions. These aggregates are then used to annotate the  $G_T$  graph nodes. Once the final  $G_T$  graph is constructed, the “Initiation Selector” also computes the set of prefetch initiation sites,  $P$ , using the algorithm from Figure 14.

Two other modules in Figure 17, the “In-liner” and the “Layout Extractor,” perform the procedure in-lining and data structure layout extraction, respectively, described in Section 6.2. These modules are implemented in SUIF, and are the largest modules in our prototype compiler. Notice, the “Profiler” output is *not* directly fed into these modules. In our compiler, cache-miss profiles are used to select prefetch initiation sites only. Our SUIF passes do not use cache-miss profiles to guide data structure layout extraction, and instead, extracts descriptors for *all* memory references identified in the code fragments. While it is relatively easy to map cache-miss load PCs to control structures such as loops and functions, it is more difficult to map them to individual memory references because the line number information used to perform the mapping provides insufficient resolution (*i.e.* multiple load PCs often map to the same line number).

In addition to extracting the data structure layout information, we must also extract the traversal code work information described in Section 3.2 to complete the LDS descriptor graphs. The “Work Extractor” in Figure 17 is responsible for this step. Like the “Initiation Selector,” the “Work Extractor” is a binary analyzer implemented in C. It constructs a control flow graph from the original program binary, and counts the number of instructions in each basic block. For each loop and recursive function in the program, the work and offset parameters illustrated in Figure 5 are extracted by aggregating the instruction counts from the appropriate basic blocks. Along with the data structure layout information from the “Layout Extractor,” these work parameters are fed

to the “Scheduler.” This module implements the scheduling algorithm presented in Sections 4.1 and 4.3, and computes the prefetch scheduling parameters required by our prefetch engine.

Finally, the “Instrumentor” is a Perl script that performs the software instrumentation necessary for multi-chain prefetching, and creates the final C code. Two types of instrumentation are performed. First, the “Instrumentor” inserts the *INIT* and *SYNC* instructions required for prefetching, as described in Section 5. Second, the “Instrumentor” also inserts code for installing the LDS descriptors into the prefetch engine. Constant LDS descriptor parameter values are written into the prefetch engine once at the beginning of the application. The remaining LDS descriptor parameter values that are unknown at compile time are written into the prefetch engine prior to prefetch initiation, *i.e.* right before the corresponding *INIT* instruction.

## 7 Experimental Methodology

This section presents our experimental methodology. First, Section 7.1 describes our simulator infrastructure. Second, Section 7.2 presents the benchmarks and simulation windows chosen for the experiments. Third, Section 7.3 discusses the compilation of benchmark codes. Finally, Section 7.4 describes other prefetching techniques we consider in our study.

### 7.1 Multi-Chain Prefetching Simulator

We constructed a detailed event-driven simulator of the prefetch engine architecture described in Section 5 coupled with a state-of-the-art RISC processor. Our simulator uses the processor model from the SimpleScalar toolset [1] to model a 1 GHz dynamically scheduled 8-way issue superscalar with an 128-entry instruction window and a 64-entry load-store dependence queue. We assume a split 32-Kbyte instruction/32-Kbyte data 2-way set-associative write-through L1 cache with a 32-byte block size. We assume a unified 1-Mbyte 4-way set-associative write-back L2 cache with a 64-byte block size. The L1 and L2 caches have 16 and 32 miss status holding registers (MSHRs) [18], respectively, which is also the number of outstanding memory requests allowed from each cache. Access to the L1 and L2 caches costs 1 and 10 cycles, respectively. For simplicity, we do not model contention across the L1-L2 cache bus.

The simulator we built also models a memory sub-system in detail, consisting of a single memory controller connected to 64 DRAM banks. Each L2 request to the memory controller simulates several actions: queuing of the request in the memory controller, sending of the row and column addresses to the appropriate DRAM bank, and data transfer across the memory system bus. Our simulator faithfully models contention in the DRAM banks and on the memory system bus, permitting concurrent accesses as long as there are no bank or bus conflicts. We configure the memory sub-system to have a baseline memory access cost of 100 cycles, and a peak bandwidth of 6.4 Gbytes/sec. Later in Section 8.6, we consider memory sub-systems with less memory bandwidth.

Our baseline prefetch engine follows the implementation described in Section 5.4. We assume the prefetch engine contains an 128-entry AGT. If an AGT entry attempts to generate an address requiring the allocation of a new AGT entry (see Section 5.2) and all AGT entries are full, address generation for that AGT entry is suspended until an entry becomes available. Our baseline prefetch engine also has a 75-entry ADT (*i.e.* any number of LDS descriptor graphs can be accommodated as long as the aggregate number of array and linked list descriptors does not exceed 75). We sized the ADT to accommodate the maximum number of required entries for our applications; hence, our simulator does not model capacity limitations in the ADT. In addition, we model a single NACU

Olden Benchmarks					L1 Miss Rate			
Program	Input Parameters	Data Structure	FstFwd	Sim	16K	64K	256K	1M
EM3D	10K nodes, 50 iters	array of pointers	27	54	41.6	39.7	32.8	17.5
MST	1024 nodes	list of lists	183	29	27.9	27.0	25.6	19.3
Treeadd	20 levels	binary tree	143	34	3.6	3.3	3.2	3.2
Health	5 levels, 500 iters	quadtree of lists	115	47	22.6	20.0	19.0	13.3
Perimeter	11 levels	unbalanced quadtree	14	17	1.4	1.1	1.0	1.0
Bisort	250,000 numbers	binary tree	377	241	2.0	1.3	0.9	0.3
SPECint2000 Benchmarks					L1 Miss Rate			
Program	Input Parameters	FstFwd	Sim	16K	64K	256K	1M	
MCF	inp.in	2,616	124	27.2	26.0	24.4	20.1	
Twolf	ref	124	128	10.8	7.9	4.9	0.9	
Parser	2.1.dict -batch < ref.in	257	119	4.9	3.1	1.6	0.6	
Perl <sub>comp</sub>	-I./lib diffmail.pl 2 550 15 24 23 100	125	132	2.0	1.3	0.9	0.3	
Perl <sub>exec</sub>	-I./lib diffmail.pl 2 550 15 24 23 100	3,276	129	2.4	0.8	0.2	0.2	

Table 1: Benchmark summary. Columns labeled “FstFwd” and “Sim” report number of fast forwarded and simulated instructions (in millions). The last four columns report the miss rate for 16K, 64K, 256K, and 1M L1 data caches (in percent).

and a single BACU, as described in Section 5.4. Each functional unit has the ability to compute one address per cycle. Finally, we couple our prefetch engine with a 64-entry 2-Kbyte prefetch buffer. The prefetch buffer is fully associative, and uses an LRU eviction policy. We assume the prefetch buffer can satisfy a processor request in 1 cycle. Each prefetch buffer entry effectively serves as an MSHR, so the prefetch engine can issue up to 64 outstanding memory requests at any time. In the event that all 64 entries are waiting on outstanding memory requests, new prefetch requests must stall until an entry becomes available. Our simulator models contention for the L1 data cache port and prefetch buffer port between the prefetch engine and the CPU, giving priority to CPU accesses.

## 7.2 Benchmarks

Our experimental evaluation uses applications from both the Olden [29] and SPECint2000 benchmark suites. Table 1 lists the applications, their input parameters, and the simulation windows used. In addition, the last four columns of Table 1 report the percent miss rate for 16K, 64K, 256K, and 1M L1 data caches from the simulation window to quantify the working set sizes of these applications. The simulation windows are defined by the columns labeled “FstFwd” and “Sim,” which report the number of fast forwarded and simulated instructions in millions, respectively, starting from the beginning of each program. The rest of this section describes how we select these simulation windows.

For Olden, all of the benchmarks consist of an initialization phase followed by a single main computation loop (or recursive function call) that accounts for all of the computation. Through code inspection, we identify the initialization code, and fast forward through this portion of the benchmark in our simulation windows. Then, we simulate the main computation to completion. One exception is Health. In addition to an initialization phase, Health also builds its primary data structure through repeated calls to `sim()` from its main computation loop. For Health, we fast forward through the first 400 calls to `sim()` to build up the data structure, and then we simulate the next 100 calls.

App	Initialization	Main Computation	Time	Total Iter	Sim Iter
MCF	<code>primal_start_artificial()</code>	<code>primal_net_simplex()</code>	58%	361,717	2,000
Twolf	<code>read_cell()</code>	<code>uloop()</code>	95%	522,568	35,000
Parser	<code>read_dictionary()</code>	<code>batch_process()</code>	90%	7,761	4
Perl	<code>perl_construct()</code>	<code>yyparse()</code>	1%	136,098	45,000
		<code>runops_standard()</code>	99%	115,249,512	500,000

Table 2: Source code level information used to select simulation windows for the SPECint2000 benchmarks. The columns labeled “Initialization” and “Main Computation” list the routines responsible for program initialization and most of the main computation, respectively. The column labeled “Time” reports the percentage of execution time spent in the computation routines. The last two columns report the total iterations and simulated iterations of the primary loop from each computation routine.

Compared to Olden, the SPECint2000 benchmarks are far more complex, making it difficult to pick representative simulation regions. We ran each SPECint2000 benchmark natively on an UltraSPARC workstation from beginning to end under gprof [10]. Using the gprof profiles and code inspection, we were able to identify the routines responsible for both program initialization and most of the main computation. Through code inspection of the main computation routines, we found that in all cases there is a large primary loop which accounts for all of the routines’ execution time (we note these primary loops are quite different from the computation loops in Olden; they call several procedures, each of which also contain multiple loops). Our simulation windows for the SPECint2000 benchmarks fast forward through the initialization routines as well as all code leading up to the primary loops, and then simulates each primary loop for several iterations.

Table 2 reports the source code level information obtained through gprof and code inspection that we used to select the simulation windows for SPECint2000. The columns labeled “Initialization” and “Main Computation” list the names of the initialization and main computation routines discovered via gprof, respectively, and the column labeled “Time” reports the fraction of time each benchmark spends in the computation routines as a percentage of the benchmark’s *entire execution time* (the fraction of time spent in the initialization routines is less than 1% for all the benchmarks). Finally, the last two columns show the total number of iterations executed in the primary loops from each main computation routine and the number of iterations we simulate in our simulation windows from Table 1, labeled “Total Iter” and “Sim Iter,” respectively. Although Table 2 shows we only simulate 4 iterations for Parser, we reiterate these are top-level loops that represent a significant portion of the overall benchmark. As Table 1 shows, Parser’s 4 iterations account for roughly 120M instructions.

In Table 1, two simulation regions are reported for Perl. Perl executes both a compile phase and an execute phase corresponding to the `yyparse()` and `runops_standard()` main computation routines listed in Table 2. Although the execute phase runs much longer than the compile phase for the inputs we use (as indicated by the “Time” column in Table 2), we believe the compile phase can still be important, particularly for perl scripts that run for short amounts of time. Consequently, we simulate both phases and treat them as separate applications, called “Perl<sub>comp</sub>” and “Perl<sub>exec</sub>.”

### 7.3 Compiling

For the majority of our experiments, we use our prototype compiler, described in Section 6.3, to instrument our benchmarks with multi-chain prefetching. In this case, all steps required for

App	#Graph	Avg Depth	#Desc	Recurrent Descriptor		Composition Type		
				Array	Linked List	Nesting		Recurse
						w/o Ind	w/ Ind	
EM3D	2	3.0(3)	13(7)	2(1)	0(0)	5(3)	6(4)	0(0)
MST	1	6.0(6)	10(10)	0(0)	2(2)	0(0)	9(9)	0(0)
Treeadd	1	2.0(2)	2(2)	0(0)	0(0)	0(0)	1(1)	1(1)
Health	1	5.0(5)	24(24)	1(1)	7(7)	0(0)	23(23)	1(1)
Perimeter	1	2.0(2)	2(2)	0(0)	0(0)	0(0)	1(1)	1(1)
Bisort	1	2.0(2)	2(2)	0(0)	0(0)	0(0)	1(1)	1(1)
MCF	2	3.0(3)	9(5)	1(1)	1(1)	1(1)	6(4)	0(0)
Twolf	6	4.7(6)	74(18)	2(1)	9(3)	4(2)	64(15)	0(0)
Parser	11	3.6(5)	49(10)	2(1)	16(4)	0(0)	38(9)	0(0)
Perl <sub>comp</sub>	1	3.0(3)	3(3)	0(0)	1(1)	0(0)	2(2)	0(0)
Perl <sub>exec</sub>	1	3.0(3)	4(4)	0(0)	1(1)	0(0)	3(3)	0(0)

Table 3: LDS descriptor summary. The number of descriptor graphs in each application is presented in the column labeled “#Graph.” The average graph depth and the total number of descriptors are presented in the columns labeled “Avg Depth” and “#Desc,” respectively. The number of descriptors by type and composition is presented in the remaining columns. Values in parenthesis report information for the most complex graph from each application.

the instrumentation, including the profiling runs necessary to drive our prototype compiler, are performed automatically and require no manual intervention. For a few experiments, we also manually instrumented our benchmarks with multi-chain prefetching. In this case, we followed the methodology for inserting instrumentation by hand used in our early work on multi-chain prefetching [17]. Since our instrumentation is performed at the source-code level (for both compiler and manual approaches), we must compile the instrumented source codes into binaries before running them on our architectural simulator. For this purpose, we use the `gcc` cross compiler provided by the SimpleScalar toolset. (When compiling with `gcc`, we use the “-O2” optimization level, which turns on all supported optimizations except for function in-lining).

To provide insight into the nature of our compiler-generated instrumentation, Table 3 shows the LDS descriptor graphs extracted from the benchmarks by our prototype compiler. For each benchmark, Table 3 reports the number of descriptor graphs, labeled “# Graph,” the average depth of each graph, labeled “Avg Depth,” and the total number of descriptors across all graphs, labeled “# Desc.” Two columns labeled “Recurrent Descriptor” show the number of recurrent descriptors by the type (array versus linked list). The last 3 columns break down the total number of descriptors into composition type (whether the descriptor is composed using nesting without indirection, nesting with indirection, or recursion). Finally, the values in parenthesis in Table 3 report the same statistics, but for the most complex descriptor graph in the benchmark.

## 7.4 Other Prefetching Techniques

Our experimental evaluation includes an extensive comparison of multi-chain prefetching against three existing prefetching techniques. The first two are jump pointer techniques, while the third is an all-hardware prediction-based technique. The two jump pointer techniques we consider are jump pointer prefetching [21], also known as “software full jumping” [31], and prefetch arrays [14].

Of the jump pointer techniques studied in [31], multi-chain prefetching is actually closest to “cooperative chain jumping,” a hybrid hardware/software jump pointer technique, rather than software full jumping (see discussion in Section 9). Although an ideal comparison would pit multi-chain prefetching versus cooperative chain jumping, a full simulated implementation of that competing technique is beyond the scope of this work. As a compromise, we implement the less advanced software full jumping technique. Since we do not have compilers that can implement either software full jumping or prefetch arrays automatically (nor do such compilers exist to our knowledge), we instrument them by hand following the algorithms presented in [21] and [14].

The third prefetching technique we consider is Predictor-Directed Stream Buffers (PSB) [33], an all-hardware prediction-based technique. PSB employs a stride-filtered markov (SFM) predictor to predict L1 cache misses. Predictions made by the SFM predictor subsequently guide multiple stream buffers that prefetch ahead of the CPU. Similar to conventional stream buffers, the SFM predictor relies on a stride predictor to efficiently prefetch striding memory access patterns. In addition, the SFM predictor also relies on a markov predictor [12] to form correlations between non-striding cache-miss addresses, thus enabling prefetching of arbitrary memory access patterns.

We built a PSB model for our simulator that supports a memory interface identical to our prefetch engine model. Hence, the PSB model can be “plugged” into the same processor and memory system models described in Section 7.1, enabling a meaningful comparison between PSB and multi-chain prefetching. We faithfully simulate every aspect of the SFM predictor and stream buffers proposed in [33]. Specifically, we simulate an SFM predictor consisting of a 256-entry stride predictor coupled with a 2K-entry 1st-order markov predictor. In addition, we simulate 8 stream buffers, each containing 8 entries. For stream buffer allocation, we use the confidence allocation with priority scheduling policy [33].

Our PSB configuration is effectively identical to the best configuration studied in [33]. The only difference is our stream buffers contain 8 entries, whereas the original PSB uses only 4 entries. We employ deeper stream buffers because they provide higher performance for our benchmarks. The deeper stream buffers also make our PSB similar to our prefetch engine in terms of hardware cost. Together, the 8 stream buffers have a 2 Kbyte capacity, which is identical to our prefetch buffer. For other hardware structures, there is some difference in cost: our PSB’s markov prediction table requires 4 Kbytes,<sup>5</sup> which is roughly double the size of our ADT and AGT combined. Overall, however, the PSB and the prefetch engine use similar amounts of hardware. (In addition to this “baseline PSB,” we also study more aggressive PSBs, which we will explain later).

## 8 Results

In this section, we conduct a detailed evaluation of multi-chain prefetching. First, we present the main results for multi-chain prefetching, and compare it against jump pointer techniques and Predictor-Directed Stream Buffers in Sections 8.1 and 8.2. Then, Section 8.3 studies a limitation of multi-chain prefetching, called *early prefetch arrival*. These 3 sections consider compiler-instrumented multi-chain prefetching only. In Section 8.4, we compare the performance of compiler- and manually-instrumented versions of our benchmarks to evaluate the quality of our compiler instrumentation. Next, we perform two other studies that evaluate the sensitivity of multi-chain prefetching performance to different architectural parameters: Section 8.5 varies prefetch engine

---

<sup>5</sup>Note, this only includes the data entries in the markov prediction table, and does not account for the tag entries. It also does not account for the stride prediction table. We do not include these structures since their sizes are not given in [33].

parameters, and Section 8.6 varies memory bandwidth parameters. Finally, Section 8.7 conducts a preliminary investigation of speculation to further improve multi-chain prefetching performance, and Section 8.8 closes by discussing the limitations of our evaluation.

## 8.1 Multi-Chain Prefetching and Jump Pointer Techniques

Figure 18 presents the results of multi-chain prefetching for our applications, and compares multi-chain prefetching against jump pointer techniques. 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.” As mentioned earlier, we use our prototype compiler to instrument multi-chain prefetching for the “MC” experiments in this section. For applications that traverse linked lists, we also report the execution time of prefetch arrays in combination with jump pointer prefetching, labeled “PA” (the other applications do not benefit from prefetch arrays, so we do not instrument them). Each bar in Figure 18 has been broken down into four components: time spent executing useful instructions, time spent executing prefetch-related instructions, and time spent stalled on instruction and data memory accesses, labeled “Busy,” “Overhead,” “I-Mem,” and “D-Mem,” respectively. 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 by as much as 74% and by 40% on average for the Olden benchmarks, and by as much as 8% and by 3% on average for the SPECint2000 benchmarks. Multi-chain prefetching provides a performance boost for all applications except for *Perl<sub>comp</sub>* and *Perl<sub>exec</sub>*, where it degrades performance by 1%. Comparing the MC bars versus the JP and PA bars, multi-chain prefetching outperforms jump pointer prefetching and prefetch arrays, reducing execution time by as much as 67% and by 34% on average for the Olden benchmarks, and by as much as 27% and by 11% on average for the SPECint2000 benchmarks. Multi-chain prefetching achieves higher performance in all but two applications, MCF and *Perl<sub>exec</sub>*, where its performance is 3% and 1% lower, respectively.

Several factors contribute to multi-chain prefetching’s performance advantage compared to jump pointer prefetching and prefetch arrays. In the rest of this section, we examine three benefits of multi-chain prefetching: low software overhead, high cache-miss coverage, and low memory overhead. Later, in Section 8.3, we will study one limitation of multi-chain prefetching: early prefetch arrival.

### 8.1.1 Software Overhead

Multi-chain prefetching incurs noticeably lower software overhead as compared to jump pointer prefetching and prefetch arrays for EM3D, MST, Health, MCF, Twolf, Parser, and *Perl<sub>comp</sub>*. For MST and Parser, jump pointer prefetching and prefetch arrays suffer high jump pointer creation overhead. On the first traversal of an LDS, jump pointer prefetching and prefetch arrays must create pointers for prefetching subsequent traversals; consequently, applications that perform a small number of LDS traversals spend a large fraction of time in prefetch pointer creation code. In MST and Parser, the linked list structures containing jump pointers and prefetch arrays are traversed 4 times and 10 times on average, respectively, resulting in overhead that costs 37% (for MST) and 29% (for Parser) as much as the traversal code itself. In addition to prefetch pointer creation overhead, jump pointer prefetching and prefetch arrays also suffer prefetch 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

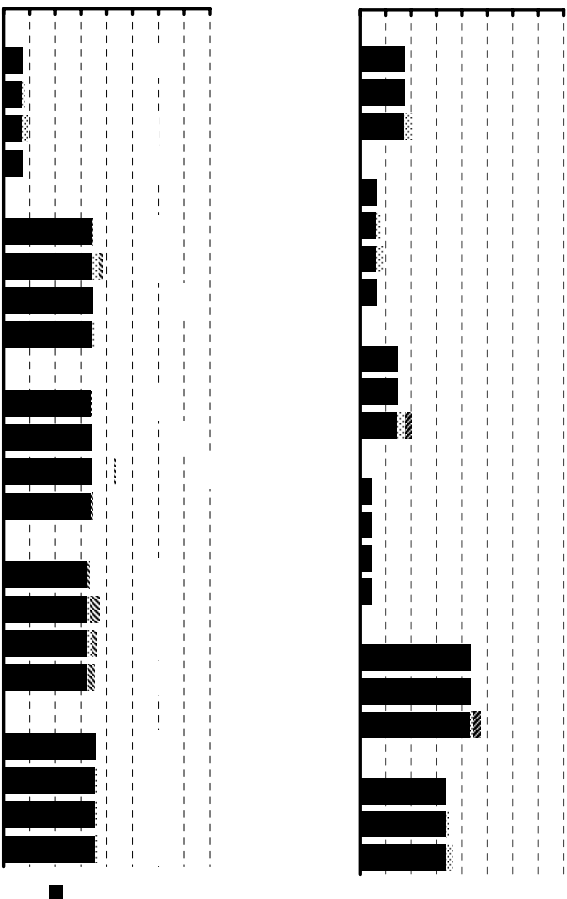


Figure 18: Execution time for no prefetching (NP), jump pointer prefetching (JP), prefetch arrays (PA), and compiler-instrumented multi-chain prefetching (MC). Each execution time bar has been broken down into useful cycles (Busy), prefetch-related cycles (Overhead), I-cache stalls (I-Mem), and D-cache stalls (D-Mem).

and delete operations. In Health, jump pointer fix-up code is responsible for most of the 129% increase in the traversal code cost. Since multi-chain prefetching only uses natural pointers for prefetching, it does not suffer any prefetch pointer creation or management overheads.

The jump pointer prefetching and prefetch array versions of EMD and MCF suffer high prefetch instruction overhead. Jump pointer prefetching and prefetch arrays insert 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). In EMD and MCF, the traversal loops are inexpensive, hence the added code in jump pointer prefetching and prefetch arrays dilates the loop cost by 84% and 23%, respectively. The particularly high overhead in EMD results in a net performance degradation of 9%. Prefetch instructions also contribute to the software overheads visible in MST, Health, and the other three SPECint2000 benchmarks. Finally, Twolf and *Perlcomp* suffer increased I-cache stalls. Due to the large instruction footprints of these applications, the code expansion caused by prefetch instrumentation results in higher I-cache miss rates. Since multi-chain prefetching does not require as much software instrumentation in these benchmarks, the impact on I-cache performance is not as significant.

Although multi-chain prefetching suffers lower software overhead than jump pointer techniques overall, it still incurs noticeable overhead in EMD, Tread, Perimeter, and Bisort. (In fact, except for EMD, the software overhead in these applications is higher for multi-chain prefetching than for the jump pointer techniques). This software overhead is due to the *INIT* and *SYNC* instructions as well as the LDS descriptor installation code inserted at each prefetch initiation site. Later, in Section 8.4, we will discuss these sources of overhead in detail.

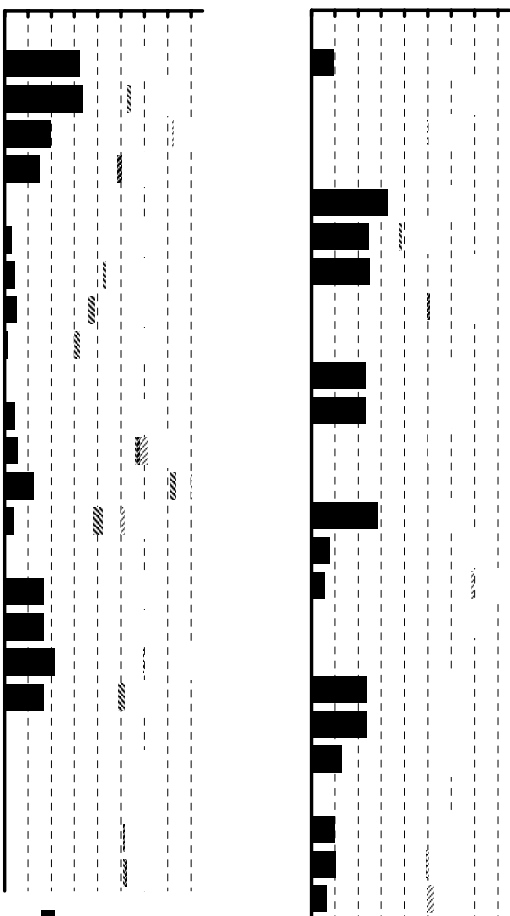


Figure 19: Cache miss breakdown for compiler-instrumented multi-chain prefetching and jump pointer techniques. Each bar has been broken down into misses to main memory (Mem), hits in the L2 (L2-Hit), partially covered misses (Partial), fully covered misses (Full), and inaccurate prefetches (Inacc).

### 8.1.2 Coverage

To further compare multi-chain prefetching and jump pointer techniques, Figure 19 shows a breakdown of cache misses for the experiments in Figure 18. The NP bars in Figure 19 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, PA, and MC bars show the same two components, but in addition show those cache misses that are fully covered and partially covered by prefetching, labeled “Full” and “Partial,” respectively. Figure 19 also shows inaccurate prefetches, labeled “Inacc.” Inaccurate prefetches fetch data that are never accessed by the processor. All bars are normalized against the NP bar for each application.

Multi-chain prefetching achieves higher cache miss coverage for Treadd, Perimeter, and Bisort due to first-traversal prefetching. (First-traversal prefetching also benefits MST, but we will explain this later in Section 8.3). In multi-chain prefetching, all LDS traversals can be prefetched. Jump pointer prefetching and prefetch arrays, however, are ineffective on the first traversal because they must create the prefetch pointers before they can perform prefetching. For Treadd 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 92%, 59%, and 34% of the original cache misses into prefetch buffer hits for Treadd, Perimeter, and Bisort, respectively, as shown in Figure 19. Figure 18 shows this increased coverage reduces execution time by 60%, 5%, and 1% for these three applications.

Figure 19 also shows the importance of prefetching early link nodes. MST and the four SPECint2000 benchmarks predominantly traverse short linked lists. In jump pointer prefetching, the first  $PD$  (prefetch distance) link nodes are not prefetched because there are no jump pointers

that point to these early nodes. However, both prefetch arrays and multi-chain prefetching are capable of prefetching all link nodes in a pointer chain; consequently, they enjoy much higher cache miss coverage typically on applications that traverse short linked lists. This explains the performance advantage of prefetch arrays and multi-chain prefetching over jump pointer prefetching for MST in Figure 18. It also explains the performance advantage of multi-chain prefetching over jump pointer prefetching for Twolf.

### 8.1.3 Memory Overhead

In addition to software overhead for creating and managing prefetch pointers as discussed in Section 8.1.1, jump pointer prefetching and prefetch arrays also incur memory overhead to store the prefetch pointers. This increases the working set of the application and contributes additional cache misses. Figure 19 shows that for Health, MCF, Twolf, Parser, and Perl<sub>comp</sub>, the total number of cache misses incurred by prefetch arrays compared to no prefetching has increased by 39%, 45%, 10%, 68%, and 42%, respectively.

The effect of memory overhead is most pronounced in Parser. This application traverses several hash tables consisting of arrays of short linked lists. Prefetch arrays inserts extra pointers into the hash table arrays to point to the link elements in each hash bucket. Unfortunately, the hash array elements are extremely small, so the prefetch arrays significantly enlarge each hash array, often doubling or tripling its size. Figure 19 shows the accesses to the prefetch arrays appear as additional uncovered cache misses. In Parser, the increase in uncovered misses outnumber the covered misses achieved by prefetching. Consequently, Figure 18 shows a net performance degradation of 55% for Parser due to prefetch arrays. For the same reasons, Twolf and Perl<sub>comp</sub> experience performance degradations of 24% and 6%, respectively. In contrast, multi-chain prefetching does not incur memory overhead since it does not use prefetch pointers. Figure 18 shows multi-chain prefetching achieves a gain of 8% and 6% for Twolf and Parser, respectively, and only suffers a 1% performance degradation for Perl<sub>comp</sub>.

In Health and MCF, the memory overhead is primarily due to jump pointers rather than prefetch arrays. Since the jump pointers themselves can be prefetched along with the link nodes, the additional cache misses due to memory overhead can be covered via prefetching. Consequently, memory overhead does not result in performance degradation. In fact for MCF, jump pointer prefetching and prefetch arrays outperform multi-chain prefetching slightly, as shown in Figure 18.

## 8.2 Multi-Chain Prefetching and Predictor-Directed Stream Buffers

Figure 20 compares multi-chain prefetching against PSB. The first and last bars in Figure 20, labeled “NP” and “MC,” report the execution time without prefetching and with multi-chain prefetching, respectively, and are identical to the corresponding bars in Figure 18. (As was the case in Section 8.1, our prototype compiler instrumented multi-chain prefetching for all the “MC” experiments in this section). The bars labeled “2K1” report the execution time for the baseline PSB configuration consisting of an SFM predictor with a 2K-entry 1st-order markov table, as described in Section 7.4. The remaining bars, labeled “I1” and “I4,” will be discussed later. Each bar has been broken down into the same four components as those in Figure 18.

Comparing the MC bars versus the 2K1 bars, multi-chain prefetching outperforms PSB on 7 out of 11 applications (EM3D, MST, Treeadd, Health, Bisort, MCF, and Twolf), while PSB outperforms multi-chain prefetching on 4 applications (Perimeter, Parser, Perl<sub>comp</sub>, and Perl<sub>exec</sub>).

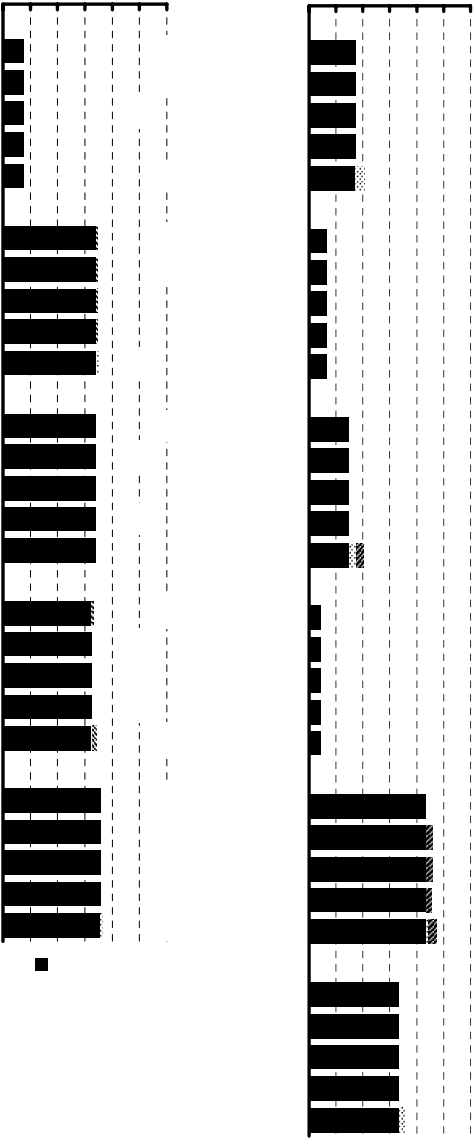


Figure 20: Execution time for no prefetching (NP), baseline PSB (2K1), PSB with an infinite 1st-order markov table (I1), PSB with an infinite 4th-order markov table (I4), and compiler-instrumented multi-chain prefetching (MC). Each execution time bar has been broken down into useful cycles (Busy), prefetch-related cycles (Overhead), I-cache stalls (I-Mem), and D-cache stalls (D-Mem).

For the Olden benchmarks, multi-chain prefetching reduces execution time compared to PSB by 27% on average. For the SPECint2000 benchmarks, however, PSB reduces execution time compared to multi-chain prefetching by 0.2% on average. Across all 11 applications, multi-chain prefetching provides higher performance, outperforming PSB by 14%.

In the remainder of this section, we study the performance differential between multi-chain prefetching and PSB in detail. We note this comparison is qualitatively different from the comparison between multi-chain prefetching and jump pointer techniques conducted in the previous section. Because jump pointer techniques rely on software instrumentation to create memory parallelism, software and memory overheads play a crucial role in our analysis in Section 8.1. In contrast, PSB does not incur any instrumentation-related overheads since it is an all-hardware technique. Instead, the key issue affecting PSB performance is *hardware predictor accuracy*. Hence, our study focuses on the two types of predictors in PSB, stride and markov, evaluates their accuracy, and studies their impact on overall performance in comparison to multi-chain prefetching.

### 8.2.1 Stride Predictor Performance

To provide insight into PSB’s behavior, Figure 21 shows a breakdown of cache misses using the same format as Figure 19. Looking at the 2K1 bars in Figure 21, we see the baseline PSB configuration converts a significant number of cache misses into fully covered misses for EM3D, Treadd, Perimeter, Parser, Perl<sub>comp</sub>, and Perl<sub>exec</sub>. In particular, for Perimeter, Parser, Perl<sub>comp</sub>, and Perl<sub>exec</sub>, PSB achieves higher cache miss coverage than multi-chain prefetching, which explains its performance advantage over multi-chain prefetching on these 4 applications in Figure 20. Interestingly, the baseline PSB configuration covers most of its cache misses through stride prefetching. To illustrate this, Table 4 breaks down the total covered cache misses into the percentage whose predictions come

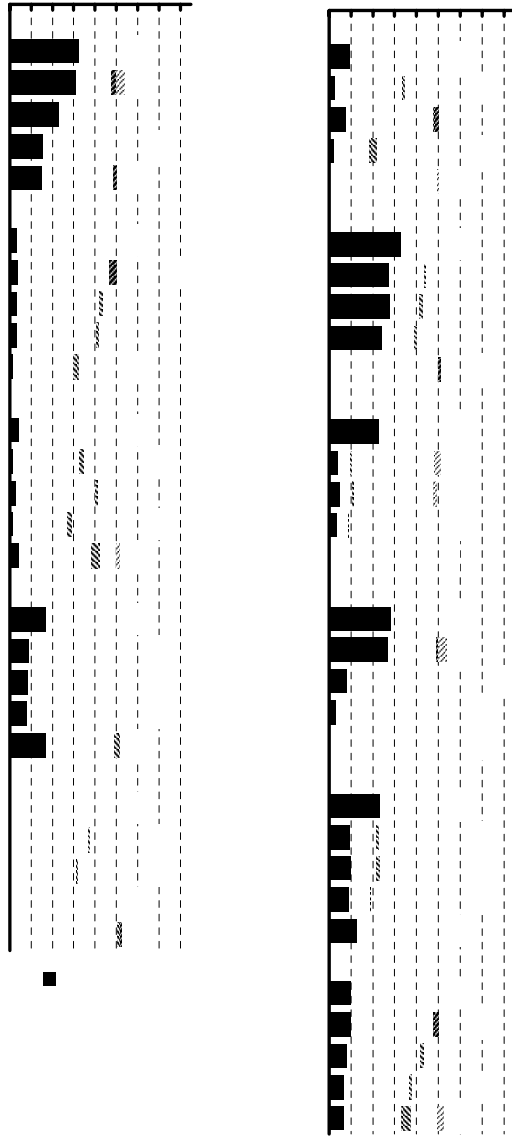


Figure 21: Cache miss breakdown for compiler-instrumented multi-chain prefetching and PSB. Each bar has been broken down into misses to main memory (Mem), hits in the L2 (L2-Hit), partially covered misses (Partial), fully covered misses (Full), and inaccurate prefetches (Inacc).

from the stride predictor versus the percentage whose predictions come from the markov predictor. With the exception of *Perfexec*, the 2K1 rows in Table 4 clearly show stride prediction is responsible for the majority of covered misses in the baseline PSB configuration.

This result is surprising. Since our applications are pointer-intensive, we did not expect a significant number of striding memory references. Upon closer examination, we discovered the striding reference patterns are due to pointer-chasing loads that reference link nodes laid out linearly in memory. This occurs with some frequency because the memory allocator, *malloc*, tends to allocate objects in a linear fashion. Hence, pointer-chasing loads exhibit striding with high likelihood whenever link nodes are visited in the order of their allocation. Such “dynamically striding” pointer-chasing loads benefit PSB because striding patterns are typically easier to predict accurately than non-striding patterns, and also because stride prediction state can be expressed much more compactly than markov prediction state. In contrast, execution-based prefetching techniques that rely on code analysis, such as multi-chain prefetching, are oblivious to dynamically striding loads and thus cannot exploit them because such loads appear to be pointer-chasing in the application code.

### 8.2.2 Markov Predictor Performance

Despite the effectiveness of stride prefetching, Figure 21 shows the 2K1 configuration of PSB achieves fewer fully and partially covered cache misses compared to multi-chain prefetching on 7 applications, leading to multi-chain prefetching’s performance advantage for these 7 applications, as illustrated in Figure 20. The reason for PSB’s lower overall performance is poor markov prefetching. In Table 4, the 2K1 rows show the baseline PSB’s markov predictor accounts for only 19.9% of all successful prefetches on average, with 8 applications receiving fewer than 15% of their prefetches from the markov predictor. This data indicates the baseline PSB’s 2K-entry 1st-order markov predictor is insufficient to capture the important cache-miss address correlations in our applications.

		EM3D	MST	Treeadd	Health	Perimeter	Bisort
2K1	Stride	99.8%	95.6%	98.9%	92.4%	99.0%	68.2%
	Markov	0.2%	4.4%	1.1%	7.6%	1.0%	31.8%
I1	Stride	42.4%	6.1%	98.8%	0.6%	98.9%	19.7%
	Markov	57.6%	93.9%	1.2%	99.4%	1.1%	80.3%
I4	Stride	43.0%	50.8%	98.5%	0.9%	100.0%	14.5%
	Markov	57.0%	49.1%	1.5%	99.1%	0.0%	85.5%
		MCF	Twolf	Parser	Perl <sub>comp</sub>	Perl <sub>exec</sub>	Average
2K1	Stride	87.0%	57.0%	85.6%	95.6%	1.6%	<b>80.1%</b>
	Markov	13.0%	43.0%	14.4%	4.4%	98.4%	<b>19.9%</b>
I1	Stride	46.0%	2.8%	34.7%	89.9%	0.3%	<b>40.0%</b>
	Markov	54.0%	97.2%	65.3%	10.1%	99.7%	<b>60.0%</b>
I4	Stride	7.0%	17.4%	86.6%	95.8%	1.2%	<b>46.9%</b>
	Markov	93.0%	82.6%	13.4%	4.2%	98.8%	<b>53.1%</b>

Table 4: Breakdown of total covered cache misses into the percentage whose predictions come from the stride predictor versus the percentage whose predictions come from the markov predictor. Breakdowns are given for the 2K1, I1, and I4 PSB configurations.

To better understand the fundamental limitations of markov prefetching for our applications, we now study *ideal* markov predictors.

First, we quantify the impact of limited capacity on markov prefetching performance by replacing the 2K1 predictor with an infinite 1st-order markov predictor. The “I1” bars in Figures 20 and 21, and the “I1” rows in Table 4 report the performance of this ideal PSB configuration. Comparing the I1 and 2K1 rows in Table 4, we see markov prefetching increases substantially for all applications, with the infinite markov predictor accounting for 60% of successful prefetches. Due to this increase in markov prefetching, Figure 21 shows 4 applications (Health, Bisort, MCF, and Twolf) experience noticeable boosts in cache miss coverage under I1, and the same 4 applications show noticeable performance increases compared to 2K1 in Figure 20, allowing PSB to outperform multi-chain prefetching in one additional application, MCF. However, the remaining applications do not benefit under I1. Worse yet, EM3D, MST, Treeadd, and Parser experience performance degradations. As a result, multi-chain prefetching still maintains a performance advantage of 14% overall as illustrated in Figure 20, even when PSB uses an infinite markov predictor.

While the I1 predictor makes more predictions than the 2K1 predictor due to its unlimited capacity, unfortunately, the majority of the additional predictions are incorrect. Why? In many applications, each cache-missing address often has multiple immediate successors at runtime. Since a 1st-order markov predictor correlates a miss address with its immediate predecessor only, multiple successors cause aliasing, leading to poor prediction accuracy. This affects performance in two ways. First, it increases inaccurate prefetches. As shown in Figure 21, EM3D, MST, Health, Bisort, MCF, Twolf, and Parser have much larger “Inacc” components under I1 compared to MC. For bandwidth-limited applications (especially EM3D), the inaccurate prefetches degrade performance. Second, poor markov prediction accuracy can also disrupt stride prefetching. In EM3D and Parser, the markov and stride prediction streams overlap. Since the markov predictor is given priority over the stride predictor [33], markov mispredictions often interrupt a string of correct stride predictions, thus reducing cache miss coverage.

To address mispredictions arising from aliasing, we study an infinite 4th-order markov predictor. This predictor correlates each cache-missing address with its 4 sequential predecessors rather

than its immediate predecessor only. The increased context for each correlation virtually eliminates aliasing even when cache-missing addresses have the same immediate successor since the additional predecessors are usually unique. The “I4” bars in Figures 20 and 21, and the “I4” rows in Table 4 report PSB performance using an infinite 4th-order markov predictor. Similar to the I1 configuration, Table 4 shows a significant number of prefetches are due to markov predictions under the I4 configuration, 53%. Compared to I1, however, I4’s markov predictions are much more accurate. Figure 21 shows the inaccurate prefetches in the I4 bars are back down to the levels in the 2K1 bars. Furthermore, cache miss coverage is higher in I4 as compared to 2K1 for all the applications, with the exception of *Perl<sub>exec</sub>*. Figure 20 shows the reduced inaccurate prefetches and higher overall cache miss coverage permits I4 to outperform 2K1 by 10% across all 11 applications.

Comparing the MC and I4 bars in Figure 20, we see PSB performance begins to approach multi-chain prefetching performance. Multi-chain prefetching outperforms PSB in EM3D, MST, Treeadd, Health, and Twolf by 23%, while PSB outperforms multi-chain prefetching in Perimeter, Bisort, MCF, Parser, *Perl<sub>comp</sub>*, and *Perl<sub>exec</sub>* by 11%. Across all 11 applications, multi-chain prefetching maintains a small advantage of 6%. Considering PSB is a completely transparent technique whereas multi-chain prefetching requires a compiler to insert software instrumentation, this is a positive result for PSB. On the other hand, PSB requires large prediction tables to achieve comparable performance to multi-chain prefetching. For applications that benefit from the I4 predictor, we ran several simulations to determine the smallest predictor that achieves similar performance to the I4 bars in Figure 20. We found 64K- to 512K-entries are necessary to come within 20% of the I4 predictor, with Health requiring 64K-entries, MCF requiring 128K-entries, Bisort and Twolf requiring 256K-entries, and MST requiring 512K-entries. (Practically speaking, such predictors are too large for the L1 cache; however, they are feasible for integration with the L2 cache. Although we have not evaluated L2 prefetching using PSB, we expect similar results to those in Figure 20 because our aggressive processor can tolerate the L1-L2 latency in many cases.)

### 8.2.3 Limits on Predictability

We tried to improve PSB performance further, but was unable to outperform the I4 configuration. In so doing, we found 4 bottlenecks that limit the accuracy of PSB’s markov predictor. Two of these are familiar: LDS modification and first-traversal prefetching. As we saw in Section 8.1.1, LDS insert and delete operations in Health require fixup code to keep jump pointers consistent. Similarly, the markov predictor requires retraining after LDS modifications because predictor entries become outdated. In addition to Health, we also found LDS modification limits predictor accuracy in Bisort as well. In Section 8.1.2, we saw jump pointer techniques cannot perform first-traversal prefetching because jump pointers have yet to be created. Similarly, the markov predictor cannot perform first-traversal prefetching because the predictor has yet to be trained. First-traversal prefetching limits PSB performance in MST, Treeadd, and Bisort.<sup>6</sup>

The two remaining bottlenecks limiting prediction accuracy are unique to PSB. First, out-of-order execution reorders cache misses across traversals of the same LDS if multiple misses occur in the processor’s instruction window simultaneously (this happens frequently in EM3D). Reordered misses limit markov prediction accuracy, especially in the I4 predictor where each predictor lookup must match a sequence of 4 miss addresses. And finally, even when an application traverses the same

---

<sup>6</sup>First-traversal prefetching is also a factor in Perimeter; however, PSB achieves high performance for this application because of stride prefetching, as discussed in Section 8.2.1. Unlike the markov predictor which requires a full traversal of the LDS for training, the stride predictor is trained after only 3 misses. Hence, it can perform prefetching on the first traversal.

LDS multiple times, low prediction accuracy occurs if the traversals are not sufficiently similar. For example, in MST, several linked lists are traversed repeatedly. Because each list is short, stream buffers must prefetch continuously from one list to the next to achieve performance. Unfortunately, separate traversals of the same linked list terminate at different link nodes, making the combined address streams of multiple list traversals unpredictable. Similar dynamic runtime behavior reduces prediction accuracy in Twolf as well.

### 8.3 The Early Prefetch Arrival Problem

Multi-chain prefetching begins prefetching a chain of pointers prior to their traversal by the processor in order to overlap a portion of the serialized memory latency with work outside of the traversal code. Early initiation of pointer chain prefetching is fundamental to the multi-chain prefetching technique. Unfortunately, it also leads to early prefetch arrival. Prefetches issued for link nodes near the beginning of a pointer chain tend to arrive early. Until the processor references the prefetched data, these early prefetches occupy the prefetch buffer. For applications with long pointer chains or small traversal loops, the number of early prefetches can exceed the prefetch buffer capacity, causing prefetched data to be evicted before the processor has had a chance to reference the data.

This section examines the early prefetch arrival problem. First, we discuss the performance impact of early prefetch arrival. Then, we evaluate the potential for mitigating early prefetches by reducing the prefetch distance computed by the scheduling algorithm in our compiler.

#### 8.3.1 Performance Impact of Early Prefetches

The early prefetch arrival problem manifests itself as thrashing in the prefetch buffer. To see whether any of our applications experience prefetch buffer thrashing, we instrumented our simulator to count each useful prefetch (*i.e.* a prefetch for which there is a processor reference sometime after the prefetch arrives) that is evicted from the prefetch buffer before the processor has had a chance to reference it. We call these *evicted useful prefetches*. The column labeled “MC” in Table 5 reports the number of evicted useful prefetches seen by the prefetch buffer as a percentage of the total useful prefetches. Three of our applications exhibit an unusually high percentage of evicted useful prefetches, indicating prefetch buffer thrashing: EM3D, MST, and Health. Two other applications, Bisort and Parser, also exhibit the problem, though to a lesser extent.

Prefetch buffer thrashing due to early prefetch arrival reduces the number of fully covered misses. In many cases, however, a reference to an evicted block will at least be partially covered. Since prefetches that miss all the way to main memory are placed in the L2 cache en route to the prefetch buffer, the processor will normally enjoy an L2 hit for prefetches from main memory. This is because the L2 cache has considerably more capacity than the prefetch buffer, allowing it to avoid thrashing even when prefetches arrive early. Figure 19 (as well as Figure 21) confirms these observations. For MST and Health, the “MC” bars in Figure 19 show multi-chain prefetching is unable to achieve any fully covered misses due to prefetch buffer thrashing. Similarly in EM3D, many prefetches are only partially covered. (Note that Perimeter, MCF, Twolf, and Perl<sub>comp</sub> also exhibit partially covered misses, but these are due to *late* rather than early prefetches since Table 5 shows these applications do not suffer prefetch buffer thrashing).

Despite early prefetch arrival in EM3D, MST, and Health, multi-chain prefetching still outperforms jump pointer prefetching, prefetch arrays, and PSB for these applications. In EM3D, limited prefetch buffer capacity causes thrashing even for jump pointer prefetching. Since multi-chain

prefetching has lower software overhead, it achieves a 41% performance gain over jump pointer prefetching on EM3D, as shown in Figure 18. In contrast, PSB does not suffer thrashing because stream buffers perform dynamic flow control (*i.e.* prefetching is suspended when a stream buffer fills and is resumed only after some data is consumed by the processor). As Figure 21 shows, most cache misses covered by PSB in EM3D are fully covered, and under the I4 configuration, PSB achieves higher coverage than multi-chain prefetching. However, inaccurate prefetches limit PSB performance, allowing multi-chain prefetching to achieve a 23% gain over PSB on EM3D, as shown in Figure 20. Comparing prefetch arrays and multi-chain prefetching for MST, we see that prefetch arrays leaves 76% of the original misses to memory unprefetched. This is due to the inability to perform first-traversal prefetching using prefetch arrays. In contrast, multi-chain prefetching converts all of MST’s “D-Mem” component into L2 hits (these appear as partially covered misses), allowing multi-chain prefetching to achieve a 67% performance gain over prefetch arrays on MST, as shown in Figure 18. Similar to prefetch arrays, PSB also cannot perform first-traversal prefetching. In addition, as discussed in Section 8.2.3, the cache-miss stream in MST is difficult to predict due to dynamic runtime behavior. Consequently, PSB covers 4 times fewer cache misses compared to multi-chain prefetching, allowing multi-chain prefetching to achieve a 68% performance gain over PSB on MST, as shown in Figure 20.

Finally, for Health, Figure 19 shows prefetch arrays converts 49% of the original cache misses into fully covered misses, while multi-chain prefetching converts only 33% of the original misses into partially covered misses due to early prefetch arrival. As a result, prefetch arrays tolerates more memory latency than multi-chain prefetching. However, due to the large software overhead necessary to manage prefetch pointers in Health, multi-chain prefetching outperforms prefetch arrays by 35%, as shown in Figure 18. Similarly, Figure 21 shows PSB also has higher cache-miss coverage than multi-chain prefetching in Health under the I1 and I4 configurations. However, because PSB is unable to remove all of the “Mem” misses, it does not eliminate as much memory stall as multi-chain prefetching. As a result, multi-chain prefetching outperforms the I4 configuration of PSB by 19% on Health, as shown in Figure 20.

### 8.3.2 Optimal Prefetch Distance Performance

While early prefetch arrival is a fundamental limitation in multi-chain prefetching, the number of early prefetches in our experiments is excessively large due to two overly conservative assumptions made by our prototype compiler. First, as discussed in Section 4.3, our scheduling algorithm computes a bounded prefetch distance to accommodate unknown list lengths and recursion depths. The bounded prefetch distance is a worst-case prefetch distance, and thus guarantees that prefetches never arrive late. But depending on the actual size of data structures, the bounded prefetch distance may initiate prefetching of pointer chains earlier than necessary, exacerbating the early prefetch arrival problem. Second, our scheduling algorithm conservatively assumes all prefetch requests incur the full latency to physical memory (*i.e.* the “ $l$ ” parameter introduced in Section 4.1). At runtime, many prefetch requests may hit in the L1 or L2 cache, reducing the effective memory latency. Similar to bounded prefetch distances, using the full main memory latency to compute the prefetch chain schedule results in unnecessarily large prefetch distances.

To evaluate the degree to which early prefetch arrival is caused by the conservative assumptions made in our prototype compiler, we varied the prefetch distance for EM3D, MST, Health, Bisort, and Parser to identify an optimal prefetch distance that minimizes the number of early prefetches, hence reducing thrashing in the prefetch buffer. Figure 22 shows the result of this experiment. The bars labeled “MC” are the same as those from Figure 18, and the bars labeled “MCo” report the

App	MC	App	MC
EM3D	23.6 %	MCF	0.0 %
MST	54.2 %	Twolf	0.0 %
Treeadd	0.0 %	Parser	6.4 %
Health	60.6 %	Perl <sub>comp</sub>	0.0 %
Perimeter	0.0 %	Perl <sub>exec</sub>	0.0 %
Bisort	7.6 %		

Table 5: Percentage of evicted useful prefetches in the prefetch buffer using computed prefetch distances (labeled “MC”).

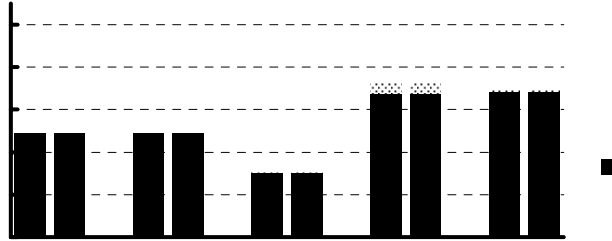


Figure 22: Multi-chain prefetching performance when prefetch distances are reduced to an optimal value determined experimentally. The “MC” and “MCo” bars show performance with the computed prefetch distance and the optimal prefetch distance, respectively. The number appearing at the base of each bar reports the prefetch distance used.

normalized execution time using the optimal prefetch distance. (Note, except for manually varying the prefetch distance to find the optimal value, both the “MC” and “MCo” bars otherwise use the instrumentation generated by our prototype compiler). The actual prefetch distance used for the most important LDS descriptor appears at the base of each bar, indicating by how much the optimal prefetch distance is smaller than the computed prefetch distance. Reducing the prefetch distance to an optimal value increases multi-chain prefetching performance by 15%, 3%, and 6% for MST, Health, and Parser, respectively. For EM3D and Bisort, there is no performance change.

Figure 22 suggests that performance increases can be achieved in some cases if a reduced prefetch distance is used instead of the conservative prefetch distance computed by our prototype compiler. However, a positive result is that the computed prefetch distance seems to work well for most applications.

## 8.4 Quality of Compiler Instrumentation

In Sections 8.1 – 8.3, we evaluated the performance of multi-chain prefetching using compiler instrumentation. An important question is how good is the instrumentation generated by our compiler? This section provides insight into this question by comparing compiler- and manually-instrumented versions of our benchmarks. Our goal is to study the performance differential between these two versions, and to identify sources of inefficiency due to limitations in our compiler.

Figure 23 reports the execution time with both compiler-instrumented multi-chain prefetching,

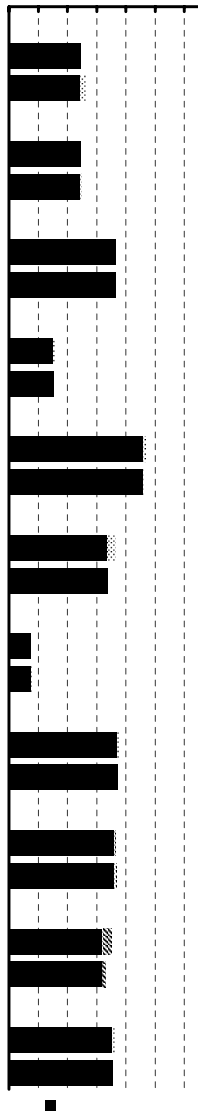


Figure 23: Execution time for compiler-instrumented (MC) and manually-instrumented (MCm) multi-chain prefetching. Each execution time bar has been broken down into useful cycles (Busy), prefetch-related cycles (Overhead), I-cache stalls (I-Mem), and D-cache stalls (D-Mem).

labeled “MC,” and manually-instrumented multi-chain prefetching, labeled “MCm.” (As discussed in Section 7.3, we follow the methodology for inserting instrumentation by hand described in [17]). Each bar has been broken down into the four components, “Busy,” “Overhead,” “I-Mem,” and “D-Mem,” and then normalized against the corresponding “MC” execution time. Except for normalization, the “MC” bars in Figure 23 are identical to those in Figures 18 and 20. Comparing the “MC” bars versus the “MCm” bars, we see manual instrumentation outperforms compiler instrumentation in 6 benchmarks (EM3D, Treadd, Bisort, MCF, Twolf, and Perl<sub>comp</sub>), reducing execution time by as much as 9%, and by 4.6% on average. For 1 benchmark (Health), compiler instrumentation outperforms manual instrumentation by 3%. And in the remaining 4 benchmarks, there is no change.

These differences arise because of discrepancies in how the compiler and manual approaches perform the following three tasks: prefetch initiation site selection, LDS descriptor graph extraction, and synchronization instrumentation. In the remainder of this section, we investigate each of these in greater detail. We begin by studying the differences in selecting prefetch initiation sites. In Bisort, our compiler selects one of the recursive functions from the program as a prefetch initiation site; however, initiating prefetching from a different recursive function yields slightly higher performance. The manual approach chooses the alternate prefetch initiation site after experimentally determining that there is a performance advantage in doing so. Our compiler misses this opportunity for achieving higher performance since it chooses prefetch initiation sites systematically based on the algorithm in Figure 14 rather than sampling potential solutions in an ad hoc fashion. In Health, one of the prefetchable LDS traversals is nested underneath multiple parents in the traversal nesting graph,  $G_T$ .<sup>7</sup> Since this LDS traversal is a pointer chasing loop, initiation of prefetching should occur from all of the traversal’s parents in  $G_T$  (see Section 6.1). Our compiler successfully selects all these prefetch initiation sites; however, the manual approach misses one of the sites due to human error.<sup>8</sup> Consequently, the compiler instrumentation covers more cache misses than the manual instrumentation, resulting in a 3% performance advantage.

Besides Bisort and Health, there are four other applications whose instrumentation is affected by prefetch initiation site selection, but in these cases, performance is not affected. In Twolf, Parser, Perl<sub>comp</sub>, and Perl<sub>exec</sub>, the manual instrumentation contains slightly more prefetch initiation sites compared to the compiler instrumentation because it uses a lower threshold for identifying cache-

<sup>7</sup>This LDS traversal occurs in the function `addlist`. It does not appear in the example kernel code from Health in Figures 13 and 15, but is similar to the `removeList` function which does.

<sup>8</sup>We discovered the error in the manual instrumentation after comparing against the compiler instrumentation. At that time, we could have fixed the problem in which case there would be no performance advantage for the compiler approach. However, we believe this honestly illustrates one of the benefits of automating multi-chain prefetching.

missing LDS traversals [17]. However, these prefetch initiation sites do not cover a significant number of additional cache misses since the cache-miss threshold used by our compiler is already quite low. Consequently, none of the “MCm” bars in Figure 23 experience a performance boost compared to the “MC” bars due to this additional prefetching.

Next, we study the differences in how the compiler and manual approaches extract LDS descriptor graphs. Overall, the compiler-extracted LDS descriptor graphs are less efficient than their manual counterparts. Specifically, we have observed four sources of inefficiency. First, our prototype compiler does not recognize write-once LDS descriptor graph parameters. As described in Section 6.3, LDS descriptor graph parameters that are not compile-time constants are installed into the prefetch engine at every prefetch initiation site, thus incurring runtime overhead repeatedly. In many cases, these dynamic parameters are written only during program initialization; hence, it is safe to install them into the prefetch engine once (after initialization). This optimization is performed in the manual approach, but not by our compiler. For two applications, Twolf and MCF, wasteful reinstallation of runtime constants into the prefetch engine accounts for the compiler instrumentation’s higher overhead compared to the manual instrumentation. In Figure 23, the effect is particularly noticeable for Twolf.

Second, our prototype compiler does not minimize initial address expressions at prefetch initiation sites. As described in Section 6.2, a dummy `init` node is added to each LDS descriptor graph (*e.g.* Figure 16g), and code is generated to install the initial address for this `init` node into the prefetch engine at runtime. In EM3D, the initial address expressions are fairly complex, so they incur noticeable runtime overhead. In the manual approach, this runtime overhead is partly mitigated by off-loading as much of the initial address computation from the instrumentation code onto the prefetch engine as possible, thus stream-lining the instrumentation code. No such optimization is attempted by our compiler; hence, the manual instrumentation achieves a 9% performance gain over the compiler instrumentation for EM3D.

Third, the compiler instrumentation incurs more I-cache misses than the manual instrumentation in `Perlcomp` due to increased cache conflicts. Both the manual and compiler approaches generate a single procedure to install LDS descriptor graph parameters that are compile-time constants into the prefetch engine at the beginning of the application. The two approaches typically place this initialization procedure in different locations, causing the code for compiler- and manually-instrumented benchmarks to be laid out differently. Normally, this discrepancy does not impact performance. However, in `Perlcomp`, the number of I-cache misses is highly sensitive to code layout due to an unusually large instruction footprint. Unfortunately, the location of the initialization procedure chosen by our compiler results in more I-cache conflict misses than the location chosen by the manual approach, causing a 3% performance degradation.

And fourth, our prototype compiler extracts descriptors for some memory references that do not incur many cache misses. Recall from Section 6.3 that our compiler uses cache-miss profiles to select prefetch initiation sites only. When extracting LDS descriptor graphs, our compiler extracts descriptors for all memory references identified inside code fragments. In contrast, the manual approach *does* use cache-miss profiles to guide LDS descriptor extraction. Consequently, the compiler-extracted LDS descriptor graphs contain significantly more descriptors than their manually-extracted counterparts. Fortunately, the additional LDS descriptors are singleton descriptors whose parameters are always compile-time constants. Furthermore, the prefetch engine resources consumed at runtime by the additional LDS descriptors do not impede the progress of critical prefetches, as we will see in Section 8.5. For these reasons, the larger LDS descriptor graphs extracted by our compiler do not degrade performance in any of our benchmarks.

Finally, we study the differences in how the compiler and manual approaches instrument synchronization. As described in Section 6.3, the “Instrumentor” module depicted in Figure 17 inserts *SYNC* instructions to synchronize the prefetch engine with the main processor. For tree traversals, our compiler inserts a *SYNC* instruction at the top of the recursive function that performs the traversal, thus issuing a *SYNC* on every recursive call. However, *SYNC*s are unnecessary for calls that traverse beyond a leaf node. This can occur if the recursive function performs each recursive call without first testing whether the current node is a leaf, and instead tests the current node pointer for a NULL value at the top of the function. In this case, the *SYNC* instruction can be moved past the NULL conditional test, avoiding *SYNC*s for calls that traverse beyond leaf nodes. This optimization is performed by the manual approach for Treeadd, Perimeter, and Bisort, but is not performed by our compiler. Hence, the manual instrumentation achieves lower overhead than the compiler instrumentation for these benchmarks, as illustrated in Figure 23.

## 8.5 Impact of Hardware Parameters

In Sections 8.1 through 8.4, we studied multi-chain prefetching assuming the baseline architecture parameters described in Section 7. This section and the next (Section 8.6) study the sensitivity of our results to variations in these parameters. In this section, we vary the size of the AGT and the prefetch buffer, two structures that dominate the hardware budget of multi-chain prefetching (see Section 5.4). Our goal is to evaluate the performance of the prefetch engine under different hardware budgets. Then later in Section 8.6, we vary the available memory bandwidth, and evaluate multi-chain prefetching under limited memory bandwidth conditions. (Note, all multi-chain prefetching experiments performed in these two sections use the compiler-instrumented versions of our benchmarks).

### 8.5.1 Varying AGT Size

Our baseline prefetch engine assumes an AGT with 128 entries. Before varying the AGT size, we first measure the number of active AGT entries in the baseline AGT to see whether our benchmarks can benefit from a larger AGT. In Table 6, we report the maximum and average number of active entries in the AGT across our benchmarks. Table 6 shows that for 8 applications (EM3D, Treeadd, Health, Perimeter, Bisort, Twolf, Perl<sub>comp</sub>, and Perl<sub>exec</sub>), the maximum number of active AGT entries, reported in the column labeled “Max AGT,” is less than the total number of AGT entries. For these applications, increasing the AGT size will not provide any benefit since the baseline AGT already provides more entries than these applications can use.

For 3 applications (MST, MCF, and Parser), the maximum number of active AGT entries is the entire AGT. Increasing the AGT size could potentially benefit these applications by reducing AGT stalls that occur when the AGT runs out of entries (see Section 7.1). To determine whether a larger AGT can improve performance, we measured the execution time of these 3 applications as the size of the AGT is increased from the baseline size of 128 entries to 512 entries. We also simulated an AGT with only 64 entries. The result of this experiment appears in Figure 24. Figure 24 shows that varying the size of the AGT has very little impact on performance. Across the entire range of AGT sizes we simulated, execution time varies by less than 1% in all 3 applications, even when the AGT is reduced to 64 entries. (We also verified that the other 8 benchmarks are similarly insensitive to variations in AGT size within the range 64-512 entries.)

The reason why varying the AGT size does not impact performance is because the average number of active AGT entries is low. The column labeled “Avg AGT” in Table 6 shows the

App	Max AGT	Avg AGT	App	Max AGT	Avg AGT
EM3D	103	17.85	MCF	128	1.63
MST	128	21.12	Twolf	38	0.73
Treeadd	55	33.19	Parser	128	3.40
Health	95	34.70	Perl <sub>comp</sub>	6	0.01
Perimeter	32	14.45	Perl <sub>exec</sub>	2	0.00
Bisort	37	9.68			

Table 6: Maximum and average AGT-entry utilization on the baseline configuration with 128 AGT entries.

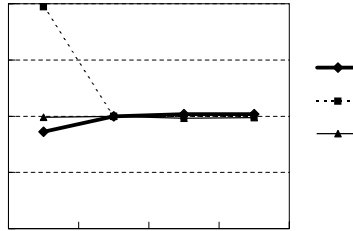


Figure 24: Execution time with varying number of AGT entries.

number of active AGT entries on average in all our benchmarks is much smaller than the number of entries available in the baseline AGT. Based on these results, we conclude that allocating more hardware to the AGT is not cost effective. Furthermore, reducing the AGT to 64 entries would cause minimal performance loss.

### 8.5.2 Varying Prefetch Buffer Size

As we did for the evaluation of AGT size variation, we first measure the number of active entries in the prefetch buffer assuming the baseline prefetch buffer size, 64 entries, before varying the number of entries. In Table 7, we report the maximum and average number of active entries in the prefetch buffer across our benchmarks. Our simulator assumes an entry in the prefetch buffer is active from the moment it enters the prefetch buffer until it leaves the prefetch buffer which occurs either on a processor reference to the entry or on an eviction, whichever comes first. The column labeled “Max Buffer” in Table 7 shows that 5 applications (Treeadd, Perimeter, Twolf, Perl<sub>comp</sub>, and Perl<sub>exec</sub>) do not make use of the full prefetch buffer, and the column labeled “Avg Buffer” shows that 1 additional application (MCF) has low average prefetch buffer usage. For these 6 applications, increasing the number of prefetch buffer entries will not improve performance. Hence, we focus on the remaining 5 applications, and measure their performance as the prefetch buffer size is varied.

Figure 25 reports the prefetch buffer size variation experiments for the 5 selected applications. Both the normalized execution time and the percentage of evicted useful prefetches are plotted as the number of prefetch buffer entries is varied. The execution times are normalized against the execution time with a 64-entry prefetch buffer, the baseline configuration, for each application. We varied the number of prefetch buffer entries between 32 and 4096, representing prefetch buffer sizes between 1 and 128 Kbytes. Although the high end in this range is unrealistic, we simulate the entire range to understand application behavior.

App	Max Buffer	Avg Buffer	App	Max Buffer	Avg Buffer
EM3D	64	59.74	MCF	64	7.77
MST	64	62.51	Twolf	51	5.29
Treeadd	36	22.45	Parser	64	26.25
Health	64	61.76	Perl <sub>comp</sub>	10	0.00
Perimeter	41	12.87	Perl <sub>exec</sub>	10	0.00
Bisort	64	22.69			

Table 7: Maximum and average prefetch buffer utilization on the baseline configuration with 64 entries (2 KBytes).

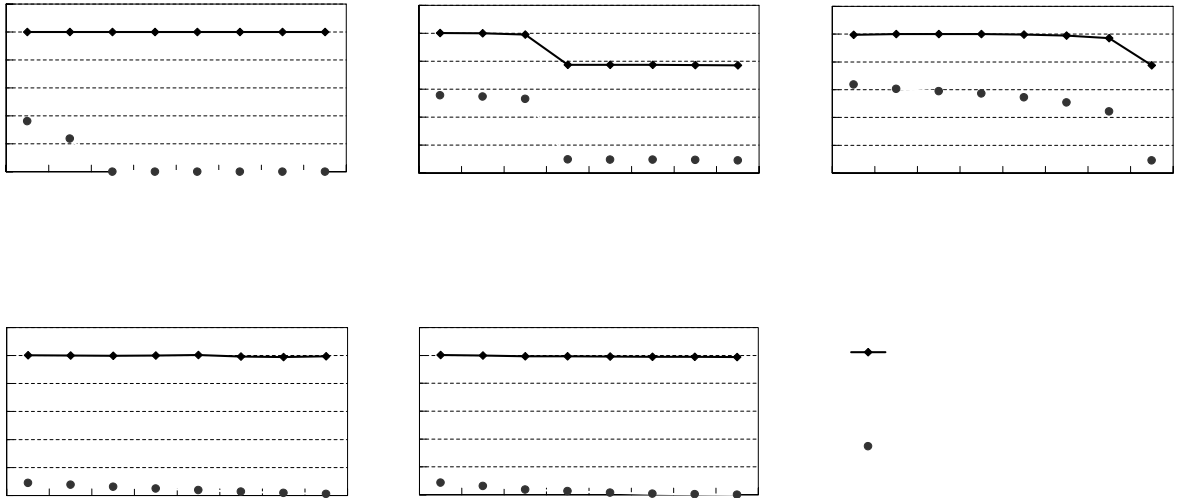


Figure 25: Normalized execution time and percentage of evicted useful prefetches as the number of prefetch buffer entries is varied between 32 and 4096 in powers of two.

MST and Health experience a significant improvement in performance with larger prefetch buffers, 23% and 24%, respectively. But EM3D, Bisort, and Parser show no gain at all. The performance gains visible in Figure 25 are due to reduced prefetch buffer thrashing. Section 8.3.2 showed that prefetch buffer thrashing can be reduced in some cases by refining the prefetch distance. Thrashing can also be reduced by increasing the prefetch buffer size. A larger prefetch buffer permits early prefetches to remain in the prefetch buffer longer, increasing the likelihood that a prefetched cache block will be accessed by the processor before it is evicted. Figure 25 supports this intuition since improvements in the execution time of MST and Health are correlated with drops in the percentage of evicted useful prefetches. Figure 25 also shows anomalous behavior for EM3D. We found that EM3D is bandwidth limited, so even though the number of evicted useful prefetches is reduced for larger prefetch buffers, performance does not improve.

Unfortunately, large prefetch buffers are necessary to fully mitigate thrashing. For EM3D, thrashing is eliminated with a 128-entry (4 Kbyte) buffer, MST requires a 256-entry (8 Kbyte) buffer, and Health requires a 1024-entry (32 Kbyte) buffer. Based on these results, we conclude

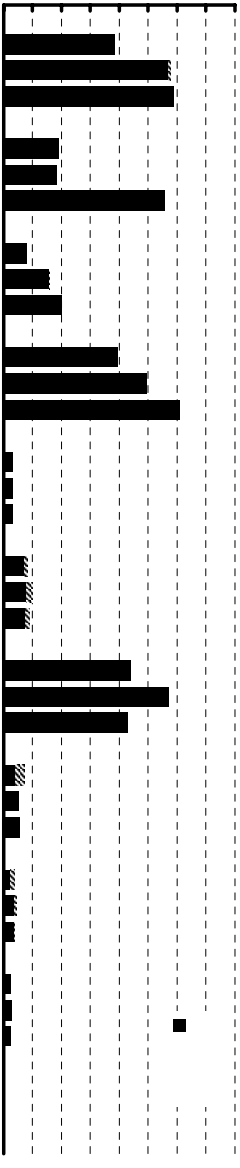


Figure 26: Memory bandwidth consumption for experiments reported in Figures 18 and 20. The “Useful” components show memory bandwidth consumed by cache blocks that are referenced by useful computation, and the “Wasteful” components show memory bandwidth consumed by all other cache blocks.

that increasing the hardware budget for the prefetch buffer can provide more performance, but unrealistically large buffers are needed to accommodate all applications.

## 8.6 Limited Memory Bandwidth Performance

Sections 8.1 through 8.5 evaluated multi-chain prefetching assuming a fixed memory sub-system model that provides an aggressive 6.4 Gbytes/sec peak memory bandwidth. In this section, we vary memory bandwidth to quantify the sensitivity of our earlier results to the available bandwidth.

To provide more insight, we first characterize the memory bandwidth consumption of our applications before studying sensitivity. Figure 26 plots the memory bandwidth consumed in each application for 3 prefetching techniques. The first is the best performing jump pointer technique, either jump pointer prefetching or prefetch arrays. The second is one of the PSB techniques: we choose the 2K1 configuration if it achieves good performance in Figure 20, otherwise we choose 14. (Our goal is to characterize memory bandwidth for the best technique. We prefer 2K1 over 14 if it has good performance since 2K1 generally has fewer inaccurate prefetches, and hence will consume less memory bandwidth). And the third is multi-chain prefetching using instrumentation generated by our prototype compiler. Each memory bandwidth consumption bar is broken down into two components. The “Useful” components report the memory bandwidth consumed fetching cache blocks that are referenced by useful computation, *i.e.* the “Busy” components in Figures 18 and 20. The memory bandwidth consumed fetching all other cache blocks is reported in the “Wasteful” components.

Figure 26 shows two important features. First, the applications consuming the most memory bandwidth tend to be the ones for which prefetching is the most effective, namely EM3D, MST, Health, and MCF. For these applications, prefetching provides a significant performance boost (see Figures 18 and 20). This performance boost increases the rate at which the application consumes data, resulting in higher memory bandwidth consumption. Second, compared to multi-chain prefetching, jump pointer and PSB techniques tend to have higher wasteful components. The jump pointer techniques in Figure 26 exhibit a significant wasteful component for EM3D, MST, Health, MCF, Twolf, and Parser. Jump pointer techniques require inserting extra pointers into linked data structures. These extra pointers must be fetched along with the applications’ “useful”

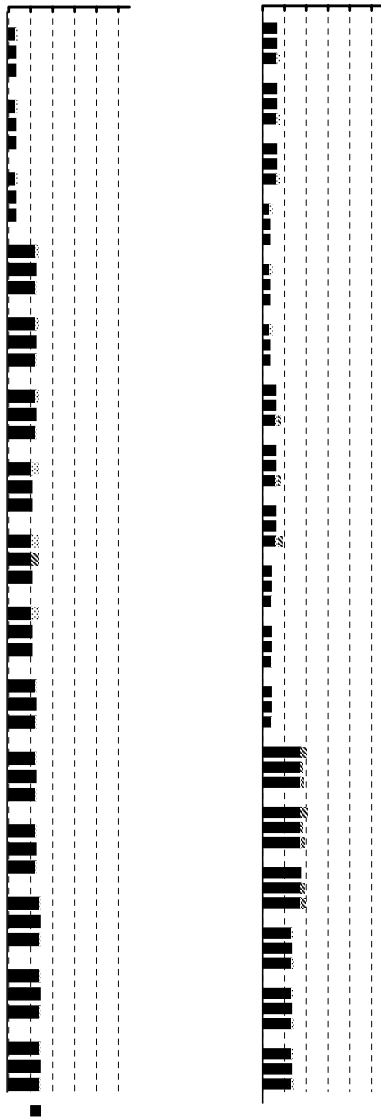


Figure 27: Normalized execution time for jump pointer techniques (JP and PA bars), PSB techniques (2K1 and I4 bars), and multi-chain prefetching (MC bars) at 6.4, 3.2, and 1.6 GBytes/sec.

data, thus increasing memory bandwidth consumption. The PSB techniques in Figure 26 exhibit a significant wasteful component for MST, Health, Bsort, MCF, Twolf, and Parser. PSB incurs inaccurate prefetches due to mispredictions. Such inaccurate prefetches increase memory bandwidth consumption as well. In contrast, multi-chain prefetching does not require extra pointers nor address prediction, so there is no wasteful component for most applications in Figure 26. Two exceptions are MST and Health. Multi-chain prefetching exhibits wasteful components for MST and Health due to linked list traversals in which link nodes are conditionally accessed. Our technique fetches each linked list entirely, resulting in wasteful fetches anytime the processor does not traverse to the end of the list.

Finally, Figure 27 presents our bandwidth sensitivity results. In Figure 27, we vary memory bandwidth from 6.4 GBytes/sec down to 1.6 GBytes/sec in powers of 2. At each memory bandwidth setting and for each application, we plot the normalized execution time for all the prefetching techniques from Figure 26.

Figure 27 shows performance generally degrades (*i.e.* execution time increases) as memory bandwidth is decreased, with EM3D, MST, Health, and MCF experiencing the largest reductions in performance. As illustrated in Figure 26, these four applications exhibit the highest memory bandwidth consumption, so they are the most sensitive to available memory bandwidth. Comparing the bars within each group across all the applications, we see that in cases where multi-chain prefetching starts out with higher performance, it maintains its performance advantage as memory bandwidth is reduced. In fact, the performance advantage widens for some applications. Recall from Figure 26 that jump pointer and PSB techniques exhibit a higher wasteful component than multi-chain prefetching. This additional memory traffic causes the performance of jump pointer and PSB techniques to degrade at a faster rate compared to multi-chain prefetching when memory bandwidth is reduced. For the jump pointer techniques, this effect is most pronounced in EM3D, MCF, Twolf, and Parser, and for the PSB techniques, it is most pronounced in MCF. In particular, for MCF, this bandwidth sensitivity discrepancy permits multi-chain prefetching to outperform prefetch arrays at 3.2 GBytes/sec, and to outperform PSB at 1.6 GBytes/sec. The only exception is Health, where the opposite occurs. As Figure 26 shows for Health, it is multi-chain prefetching

that exhibits the higher wasteful component (in comparison to PSB). Hence, multi-chain prefetching performance degrades at a faster rate, allowing PSB performance to catch up. However, even in Health, multi-chain prefetching still maintains its performance advantage over the other techniques across the entire range of memory bandwidths simulated.

## 8.7 Speculative Multi-Chain Prefetching

In this article, we have developed multi-chain prefetching and evaluated its performance for static LDS traversals only. As described in Section 2.2, handling dynamic traversals presents some difficult challenges. Unlike static traversals, the order in which pointer chains are traversed for dynamic traversals is not known until runtime. Consequently, we cannot determine the pointer-chain traversal order for dynamic traversals through static analysis of the source code, as is possible for static traversals. Without this static information, the off-line techniques developed in Sections 4 and 6 for our compiler cannot be used.

One possible approach for addressing dynamic traversals is to use *speculation*. Given a dynamic traversal, we can speculatively select the most likely pointer-chain traversal order, or when a likely ordering is not apparent, we can speculatively pursue multiple pointer chains simultaneously. Then, for each speculatively selected pointer-chain traversal order, our scheduling algorithm can be used to compute a prefetch chain schedule. Since incorrect speculation may increase the number of inaccurate prefetches and in some cases even degrade performance, speculation should not be used in an overly aggressive fashion. We call this approach *speculative multi-chain prefetching*.

While a complete investigation of speculative multi-chain prefetching is beyond the scope of this article, this section conducts a preliminary investigation of the approach. More specifically, we apply speculative multi-chain prefetching to MCF, an application for which multi-chain prefetching achieves only modest performance gains due to limited memory parallelism. Using MCF as an example, we hope to understand the potential for speculation to enable multi-chain prefetching for dynamic traversals in the future. Note, in this study, we instrument speculative multi-chain prefetching *manually*. Our intent is to conduct a preliminary study only, so we leave automation of speculative multi-chain prefetching to future work.

In MCF, a complex structure consisting of multiple backbone and rib structures is traversed. At each backbone node, there are 3 possible “next” pointers to pursue, leading to 3 different backbone nodes. A loop is used to traverse one of the next pointers until a NULL pointer is reached. This loop performs the traversal of one backbone chain statically, so our basic multi-chain prefetching technique can be used to prefetch the traversal. Unfortunately, this loop yields very little inter-chain memory parallelism. However, when this loop terminates, another backbone and rib structure is selected for traversal from a previously traversed backbone node. Since the selection of the new backbone node is performed through a data-dependent computation, the next backbone and rib traversal is not known a priori.

To enable the simultaneous traversal of multiple backbone and rib structures, we use our prefetch engine to launch prefetches down all 3 pointers speculatively at every backbone node traversed. Although we cannot guarantee that any one of these pointers will be traversed in the future, by pursuing all of them, we are guaranteed that the next selected backbone and rib structure will get prefetched. To limit the number of inaccurate prefetches caused by the mis-speculated pointers, we prefetch each chain speculatively to a depth of 5. Figure 28 evaluates the performance of this approach. In Figure 28, the bars labeled “NP,” “PA,” “I4,” “MC,” and “MCm” are identical to the corresponding bars for MCF from Figures 18, 20, and 23. (The only difference is in Figure 28, the

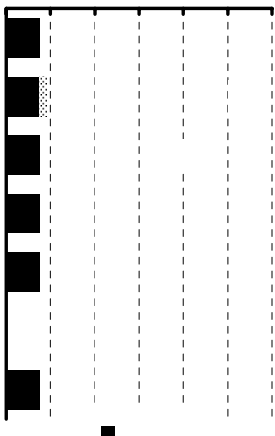


Figure 28: Preliminary performance results for speculative multi-chain prefetching on MCF. The first 5 bars are taken from Figures 18, 20, and 23. The bar labeled “SP” reports performance for speculative multi-chain prefetching.

“MCM” bar is normalized against the “NP” bar, whereas it was normalized against the “MC” bar in Figure 23). The bar labeled “SP” shows the execution time for MCF using speculative multi-chain prefetching. Comparing the “SP” bar against the “MC” and “MCM” bars, we see speculation increases performance by roughly 22% over no speculation. Comparing the “SP” and “PA” bars, we see speculative multi-chain prefetching outperforms prefetch arrays by 21%. However, PSB still holds a performance advantage. Figure 28 shows the I4 configuration of PSB outperforms speculative multi-chain prefetching by 7.5%.

## 8.8 Limitations of the Experimental Evaluation

Our experimental evaluation has presented an in-depth investigation of multi-chain prefetching. Before drawing conclusions from this study, it is important to note its limitations. The primary limitation is our choice of memory system simulation parameters, and their impact on performance gains reported in Sections 8.1, 8.2, and 8.7. Our baseline memory system assumes 6.4 Gbytes/sec. While this is realizable using current DRAM technology and an aggressive memory subsystem architecture, it represents a fairly high-end memory system. Less costly (and perhaps more common) memory systems will provide lower bandwidth, and will not achieve the same levels of performance. Hence, the results in Sections 8.1, 8.2, and 8.7 are optimistic (for all techniques, not only multi-chain prefetching), and perhaps the results in Section 8.6 are more indicative of behavior on the majority of systems. In addition, all experimental studies are limited by the choice and number of benchmarks—ours is no exception. Final conclusions about multi-chain prefetching performance can only be drawn after conducting further studies using additional applications.

## 9 Related Work

The work presented in [17] describes the early version of our multi-chain prefetching technique. Compared to our early work, this article extends multi-chain prefetching in several ways. Foremost, this article automates the technique, providing algorithms and a prototype compiler for extracting all the necessary information fully automatically. In addition, this article develops a prefetch scheduling framework that handles recursion via descriptor unrolling, and proposes limit analysis for computing bounded prefetch distances when data structure size is not known. In contrast, the framework in [17] does not handle recursion nor data structures with unknown extents. Also, this

article presents a hardware implementation of our prefetch engine, and analyzes its complexity. Finally, this article presents a more comprehensive evaluation of our technique compared to [17], providing a comparison against PSB, a detailed study of the early prefetch arrival problem, and the impact of varying architecture parameters on multi-chain prefetching performance.

Compared to the work of other researchers, this article is the first to develop a complete and systematic method for scheduling prefetch chains, and to apply it to prefetching arbitrary pointer-based data structures. However, as we stated in the Introduction to this article, exploiting inter-chain memory parallelism is not a new idea. Our work builds upon a long list of recent pointer prefetching techniques, some of which prefetch simple data structures in a multi-chain fashion.

Dependence-Based Prefetching (DBP) by Roth, Moshovos, and Sohi [30] identifies recurrent pointer-chasing loads in hardware, and prefetches them sequentially using a prefetch engine. DBP exploits inter-chain memory parallelism for simple backbone and rib traversals, but cannot do so for more complex traversals. In comparison, our technique exploits inter-chain memory parallelism for any data structure composed of lists, trees, and arrays. Furthermore, our work provides a compiler-implemented off-line algorithm for systematically scheduling prefetch chains, whereas DBP relies on hardware to perform prefetch scheduling in a greedy fashion. By identifying and scheduling recurrent loads in a compiler, our approach requires less hardware support than DBP, and more exactly times the arrival of prefetches which can reduce thrashing. On the other hand, DBP does not require compiler support.

Similar to DBP, Mehrotra and Harrison propose a hardware mechanism called the Indirect Reference Buffer (IRB) [23] to identify and prefetch recurrent loads. The IRB only identifies recurrences between dynamic instances of the same static load, so it is less general than DBP (which detects recurrences between multiple static loads). Also, the IRB does not exploit any memory parallelism.

The Push Model by Yang and Lebeck [37] performs pointer prefetching under control of the compiler through a series of prefetch engines attached to different levels of the memory hierarchy. These memory-side prefetch engines actively “push” prefetched data towards the CPU. By performing memory-side prefetching, the Push Model reduces the round-trip latency to and from memory, thus increasing the throughput of back-to-back pointer-chasing memory references. However, the Push Model still incurs serialized memory latency during pointer-chain traversal, and does not attempt to exploit memory parallelism between pointer chains.

Greedy Prefetching by Luk and Mowry [21] inserts software prefetches into LDS traversal loops to speculatively prefetch all successor nodes at each link node traversed. For linked list traversals, Greedy Prefetching does not exploit any memory parallelism because there is only one successor node. For traversals of data structures with multiple successor nodes (*e.g.* trees), Greedy Prefetching exploits some inter-chain overlap because multiple successor nodes are prefetched simultaneously (resulting in improved performance if future traversals pursue some or all of the speculatively prefetched successors). Our speculative multi-chain prefetching technique, described in Section 8.7, employs a similar form of speculation, but pursues entire pointer chains speculatively rather than single successor nodes, resulting in greater multi-chain overlap when speculation is correct.

In contrast to DBP, the IRB, the Push Model, and Greedy Prefetching which only use natural pointers to perform prefetching, several researchers have proposed jump pointer techniques. Jump Pointer prefetching by Luk and Mowry [21] (also known as History Pointer prefetching) inserts special pointers into the LDS to connect non-consecutive link elements. These jump pointers enable prefetch instructions to name link elements further down a pointer chain, thus creating memory parallelism along a single chain of pointers. Prefetch Arrays by Karlsson, Dahlgren, and

Stenstrom [14] extends Jump Pointer prefetching with additional pointers that point to the first few link elements in a linked list to permit prefetching of early nodes. This article conducts an extensive quantitative comparison of multi-chain prefetching against both Jump Pointer prefetching and Prefetch Arrays. Previous sections in this article have discussed their relative merits and shortcomings.

Roth and Sohi [31] propose several variations on Jump Pointer prefetching, and study hardware, software, and hybrid implementations of these variations. Cooperative Chain Jumping, one of their proposed techniques, exploits inter-chain memory parallelism for backbone and rib structures. In Cooperative Chain Jumping, jump pointers are created along the backbone, and a hardware prefetch engine (similar to the one in DBP) is used to prefetch each rib sequentially. The backbone jump pointers enable initiation of prefetching for multiple ribs simultaneously, thus overlapping independent rib traversals. Although Cooperative Chain Jumping exploits inter-chain memory parallelism, it does so only for backbone and rib structures, and it requires jump pointers. In contrast, our work seeks to exploit inter-chain memory parallelism for arbitrary pointer-based data structures using only natural pointers. Moreover, our work provides a systematic method for scheduling prefetch chains. Similar to DBP, Cooperative Chain Jumping performs prefetch scheduling in a greedy fashion.

Another approach for creating memory parallelism within a single chain of pointers is to linearize link node layout. When logically consecutive nodes in a linked list are laid out linearly in physical memory, the linked list effectively becomes an array and can be prefetched using stride prefetching [5]. Stoutchinn *et al* [34] observe that linked lists are often allocated on the heap in a linear fashion and apply stride prefetching in software (our results confirm Stoutchinn’s observation since we found stride prefetching is effective in several of our benchmarks). Data Linearization prefetching by Luk and Mowry [21] proposes a memory allocator that performs the linearization at node allocation time.

In addition to the techniques described thus far, other techniques exist that do not specifically target pointer-chasing references, yet are effective for LDS traversals nonetheless. One such technique is *pre-execution* [7, 8, 15, 19, 20, 24, 32, 38]. Pre-execution uses idle execution resources (for example spare hardware contexts in a simultaneous multithreading processor [36]) to run one or more helper threads in front of the main computation. The helper threads trigger cache misses on behalf of the main thread so that it executes with fewer stalls. In order for cache misses to be triggered sufficiently early, multiple data streams must be pursued simultaneously using multiple helper threads, much like our prefetch engine pursues multiple pointer chains simultaneously. Compared to multi-chain prefetching, however, pre-execution is more general since helper threads can execute arbitrary code. In contrast, our prefetch engine is designed specifically for LDS traversal.

Like pre-execution, Markov Predictors [12, 33] are another general latency tolerance technique. Markov Predictors store temporally correlated cache miss addresses in a hardware table to predict non-striding cache misses; hence, they can prefetch LDS traversals. Markov Predictors, like other hardware prediction-based techniques, have the advantage that they are transparent to software. Execution-based techniques, like multi-chain prefetching, require compiler support, but they have the potential to be more accurate since they do not suffer mispredictions due to insufficient history or limited prediction table capacity. In addition, execution-based techniques require less hardware since they do not need to store the address correlations. This article conducts an extensive quantitative comparison of multi-chain prefetching against Predictor-Directed Stream Buffers [33], a markov prediction technique, and evaluated their relative merits and shortcomings.

Unroll-and-jam by Pai and Adve [27] is a loop transformation that exploits memory parallelism.

For loops that traverse data structures with multiple independent pointer chains, like array of lists, unroll-and-jam initiates independent instances of the inner loop from separate outer loop iterations, thus exposing multiple read misses within the same instruction window. Like all loop transformations, unroll-and-jam can only be applied if it does not change the original program semantics. Prefetching techniques are not limited by such legality criteria.

Finally, preceding the literature on pointer prefetching is a large body of work on prefetching for array data structures. These include hardware prefetching techniques by Chen [5], Palacharla [28], Fu [9], and Jouppi [13], software prefetching techniques by Mowry [25, 26], Klaiber [16], and Callahan [2], and hybrid techniques by Temam [35], Chen [4], and Chiueh [6]. Since these techniques rely on the regular memory access patterns exhibited by array traversals, they are ineffective for LDS traversals.

## 10 Conclusion

This article investigates exploiting inter-chain memory parallelism for memory latency tolerance. Our technique, called *multi-chain prefetching*, issues prefetches along a single chain of pointers sequentially. However, multi-chain prefetching aggressively pursues multiple independent pointer chains simultaneously, thus exploiting the natural memory parallelism that exists between separate pointer-chasing traversals. Although the idea of overlapping chained prefetches is not new and has been demonstrated for simple backbone and rib traversals [30, 31], our work seeks to exploit inter-chain memory parallelism for arbitrary data structures consisting of lists, trees, and arrays.

Our work makes the following contributions. We introduce a framework for compactly describing static LDS traversals, providing the data layout and traversal code work information necessary for prefetching. We present an off-line algorithm for computing a prefetch schedule from the LDS descriptors that overlaps serialized cache misses across separate pointer-chain traversals. While our algorithm is designed for static LDS traversals, we also propose the use of speculation to handle dynamic LDS traversal codes as well. In addition, we present the design of a programmable prefetch engine that performs LDS traversal outside of the main CPU, and prefetches the LDS data according to the computed prefetch schedule. Next, we develop several algorithms for extracting LDS descriptors via static analysis of the application source code, and implement them in a prototype compiler, thus automating multi-chain prefetching. Finally, we conduct an in-depth experimental evaluation of compiler-instrumented multi-chain prefetching, and compare its performance against jump pointer prefetching, prefetch arrays, and predictor-directed stream buffers.

Based on our experimental results, we make several observations regarding the effectiveness of multi-chain prefetching. First, our results show multi-chain prefetching can provide significant performance gains, reducing overall execution time by 40% across six Olden benchmarks, and by 3% across four SPECint2000 benchmarks. Second, multi-chain prefetching outperforms jump pointer prefetching and prefetch arrays by 34% and 11% for the selected Olden and SPECint2000 benchmarks, respectively. The performance advantage of multi-chain prefetching over jump pointer techniques comes from its stateless nature. Multi-chain prefetching avoids software overhead for managing prefetch pointers incurred by jump pointer techniques. Multi-chain prefetching also avoids prefetch pointer storage overhead which can significantly increase the number of cache misses, particularly for applications employing small link structures. In addition, multi-chain prefetching can perform first-traversal prefetching, and can effectively tolerate the cache misses of early nodes in short lists.

Third, multi-chain prefetching achieves 27% higher performance than PSB across the selected Olden benchmarks, but PSB outperforms multi-chain prefetching by 0.2% across the selected SPECint2000 benchmarks. Overall, multi-chain prefetching achieves 16% higher performance than PSB across all 11 of our applications. PSB’s performance gains come from prefetching “dynamically striding” pointer-chasing loads. Unfortunately, markov prefetching using the baseline 2K-entry 1st-order markov predictor provides only small gains on top of stride prefetching due to insufficient capacity and predictor aliasing. With an infinite 4th-order markov predictor, PSB draws to within 6% of multi-chain prefetching across all applications; however, markov predictors with 64K- to 512K-entries are necessary to realize most of this performance gain. Based on these results, we conclude multi-chain prefetching offers a significant savings in hardware compared to PSB since it relies on the compiler rather than large prediction tables to generate the prefetch address stream.

Fourth, a fundamental limitation in multi-chain prefetching is early prefetch arrival. Since multi-chain prefetching initiates the traversal of a pointer chain prior to the traversal loop, some prefetches necessarily arrive early, in some cases causing eviction before the data is accessed by the processor. For some applications, additional performance may be achieved by reducing the prefetch distance due to the conservative nature of our compiler. However, a positive result is that the computed prefetch distance seems to work well for most of our applications.

Fifth, we find manual instrumentation of multi-chain prefetching can achieve slightly higher performance than our prototype compiler. For 6 applications, manually-instrumented multi-chain prefetching further reduces execution time by 4.6%. For 4 applications, manual instrumentation achieves no additional gain, and in 1 application, it performs slightly worse. Averaged across all 11 of our applications, the manual approach outperforms our compiler by 2.4%. The additional performance gains achieved via manual instrumentation are due to several factors (and no single factor is primarily responsible): ad hoc selection of prefetch initiation sites, identification of write-once LDS descriptor graph parameters, initial address expression minimization, cache-conscious code layout, and optimization of *SYNC* instructions for tree traversals. From this study, we conclude there is still room for improvement in our compiler algorithms, but the current algorithms seem to get most of the gain.

Finally, our results suggest that speculation can potentially uncover inter-chain memory parallelism in dynamic traversals. For MCF, we show speculation increases multi-chain prefetching performance by 22%, though PSB still outperforms speculative multi-chain prefetching by 7.5%. In future work, we plan to apply speculation more aggressively, and to identify the types of dynamic traversals amenable to speculation. However, we conclude multi-chain prefetching loses its effectiveness for highly dynamic traversals where inter-chain memory parallelism is difficult or impossible to identify.

## 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 *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.
- [3] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. February 1995.
- [4] 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.

- [5] 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.
- [6] Tzi-cker Chiueh. Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops. In *Proceedings of Supercomputing '94*, pages 488–497. ACM, November 1994.
- [7] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, Austin, TX, December 2001.
- [8] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.
- [9] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, December 1992.
- [10] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of 1982 SIGPLAN Symp. Compiler Construction*, pages 120–126, June 1982.
- [11] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE COMPUTER*, 29(12), December 1996.
- [12] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 252–263, Denver, CO, June 1997. ACM.
- [13] 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.
- [14] 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.
- [15] Dongkeun Kim and Donald Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, October 2002. ACM.
- [16] 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.
- [17] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes. In *Proceedings of the 10th Annual International Conference on Parallel Architectures and Compilation Techniques*, pages 268–279, Barcelona, Spain, September 2001. IEEE.
- [18] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pages 81–87. ACM, May 1981.
- [19] Steve S. W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoffehner, Daniel Lavery, and John P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. ACM.
- [20] Chi-Keung Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.

- [21] 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.
- [22] J. R. Lyle and D. R. Wallace. Using the unravel program slicing tool to evaluate high integrity software. In *Proceedings of 10th International Software Quality Week, USA*, May 1997.
- [23] Sharad Mehrotra and Luddy Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs. In *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996. ACM.
- [24] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *Proceedings of the International Conference on Supercomputing*, June 2001.
- [25] Todd Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems*, 16(1):55–92, February 1998.
- [26] 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.
- [27] Vijay S. Pai and Sarita Adve. Code Transformations to Improve Memory Parallelism. In *Proceedings of the International Symposium on Microarchitecture*, November 1999.
- [28] 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.
- [29] 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.
- [30] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [31] 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.
- [32] Amir Roth and Gurindar S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 191–202. ACM, January 2001.
- [33] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-Directed Stream Buffers. In *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.
- [34] Artour Stoutchinin, Jose Nelson Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain, and Alban Douillet. Speculative Pointer Prefetching of Induction Pointers. In *Compiler Construction 2001, European Joint Conferences on Theory and Practice of Software*, Genova, Italy, April 2001.
- [35] O. Temam. Streaming Prefetch. In *Proceedings of Europar’96*, Lyon, France, 1996.
- [36] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 1996 International Symposium on Computer Architecture*, Philadelphia, May 1996.
- [37] Chia-Lin Yang and Alvin R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the International Conference on Supercomputing*, May 2000.
- [38] Craig Zilles and Gurindar Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.