Appears in *Proceedings of the 10th Annual International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain*, September 2001.

Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes

Nicholas Kohout
Intel Corp.
nicholas.j.kohout@intel.com

Seungryul Choi
Computer Science Dept.
Univ. of Maryland, College Park
choi@cs.umd.edu

Dongkeun Kim, Donald Yeung
Electrical & Computer Eng. Dept.
Univ. of Maryland, College Park
{dongkeun,yeung}@eng.umd.edu

Abstract

Pointer-chasing applications tend to traverse composed 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 paper presents multi-chain prefetching*, a technique that utilizes offline analysis and a hardware prefetch engine to prefetch multiple independent pointer chains simultaneously, thus exploiting inter-chain memory parallelism for the purpose of memory latency tolerance.*

This paper makes three contributions. First, we introduce a scheduling algorithm that identifies independent pointer chains in pointer-chasing codes and computes a prefetch schedule that overlaps serialized cache misses across separate chains. Our analysis focuses on static traversals. We also propose using speculation to identify independent pointer chains in dynamic traversals. Second, we present the design of a prefetch engine that traverses pointer-based data structures and overlaps multiple pointer chains according to our scheduling algorithm. Finally, we conduct an experimental evaluation of multi-chain prefetching and compare its performance against two existing techniques, jump pointer prefetching [9] and prefetch arrays [6].

Our results show multi-chain prefetching improves execution time by 40% across six pointer-chasing kernels from the Olden benchmark suite [14], and by 8% across four SPECInt CPU2000 benchmarks. Multi-chain prefetching also outperforms jump pointer prefetching and prefetch arrays by 28% on Olden, and by 12% on SPECInt. Furthermore, speculation can enable multi-chain prefetching for some dynamic traversal codes, but our technique loses its effectiveness when the pointer-chain traversal order is unpredictable. Finally, we also show that combining multi-chain prefetching with prefetch arrays can potentially provide higher performance than either technique alone.

This research was supported in part by NSF Computer Systems Architecture grant CCR-0093110 and NSF CAREER Award CCR-0000988.

1. Introduction

Prefetching, whether using software [11, 7, 1], hardware [3, 13, 5], or hybrid [2, 4] techniques, has proven successful at hiding memory latency for applications that employ regular data structures (*e.g.* arrays). Unfortunately, these techniques are far less successful for applications that employ linked data structures (LDSs) due to the memory serialization effects associated with LDS traversal, known as the *pointer chasing problem*. The memory operations performed for array traversal can issue in parallel because individual array elements can be referenced independently. In contrast, the memory operations performed for LDS traversal must dereference a series of pointers, a purely sequential operation. The lack of *memory parallelism* prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain.

Recently, researchers have begun investigating prefetching techniques for LDS traversal, by addressing the pointer-chasing problem using two different approaches. Techniques in the first approach [17, 15, 10, 9], which we call *stateless techniques*, 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.

Techniques in the second approach [6, 16, 9], which we call *jump pointer techniques*, insert additional pointers into the LDS to connect non-consecutive link elements. These "jump pointers" allow prefetch instructions to name link elements further down the pointer chain without sequentially traversing the intermediate links, thus creating memory parallelism along a single chain of pointers. 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.

In this paper, we propose a new stateless prefetching technique

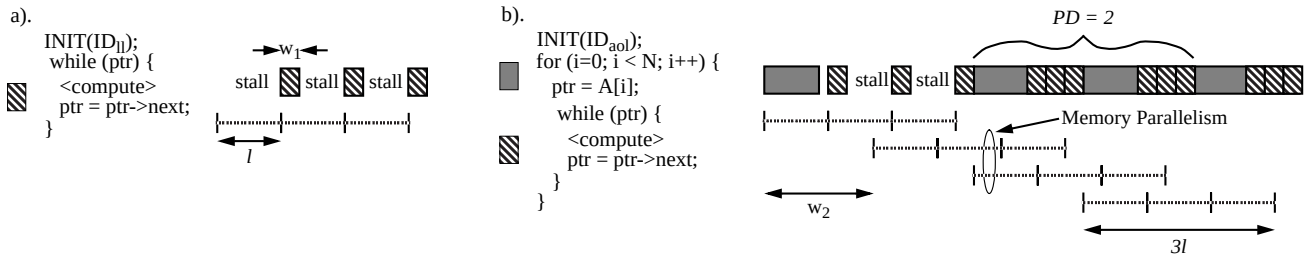

Figure 1. Traversing pointer chains using a prefetch engine. a). Traversal of a single linked list. b). Traversal of an array of lists data structure.

called *multi-chain prefetching*. Multi-chain prefetching prefetches a single chain of pointers sequentially. However, multi-chain prefetching aggressively prefetches multiple independent pointer chains simultaneously, thus exploiting the natural memory parallelism that exists between separate pointer-chasing traversals. Previous stateless techniques [16, 15] have demonstrated the potential gains of such *inter-chain* prefetching for simple "backbone and rib" structures only. Multi-chain prefetching is powerful enough to exploit inter-chain memory parallelism for any arbitrary data structure composed of lists, trees, and arrays. Due to its aggressive exploitation of inter-chain memory parallelism, multi-chain prefetching can tolerate serialized memory latency even when the 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.

Our work makes three contributions in the context of multi-chain prefetching. First, we introduce a scheduling algorithm that identifies independent pointer chains and computes a prefetch schedule that overlaps serialized cache misses across separate chains. Our analysis focuses on LDS traversals where the order in which separate chains are traversed is known a priori. We also propose and evaluate extensions to our analysis that use speculation to identify the independent pointer chains in dynamic traversal codes; however, our techniques cannot handle codes in which the pointer-chain traversals are highly unpredictable. Second, we present the design of a programmable prefetch engine that performs LDS traversal outside the main CPU, and prefetches the LDS data according to the prefetch schedule computed by our scheduling algorithm. Finally, we conduct an experimental evaluation of multi-chain prefetching, and compare it against jump pointer prefetching [16, 9] and prefetch arrays [6]. Our results show multi-chain prefetching improves execution time by 40% across six pointer-chasing kernels from the Olden benchmark suite [14], and by 8% across four SPECInt CPU2000 benchmarks. We also show multi-chain prefetching outperforms jump pointer prefetching and prefetch arrays by 28% on Olden, and by 12% on SPECInt.

The rest of this paper is organized as follows. Section 2 further explains the essence of our approach. Section 3 presents our scheduling technique. Section 4 introduces our prefetch engine, and Section 5 presents experimental results. Finally, Section 6 discusses related work, and Section 7 concludes the paper.

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, $INIT(ID_{ll})$, instructing the prefetch engine to initiate traversal of the linked list identified by the ID_{ll} 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, 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 static traversals of linked data structures in which the order of link node traversal does not depend on runtime data. For such static traversals, it is possible to determine the traversal order 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 such 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.

While 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 5.3, 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. Prefetch Chain Scheduling

This section describes how to schedule prefetch chains in multi-chain prefetching. While Section 2 describes prefetch chain scheduling for a simple array of lists example, the techniques we will present are powerful enough to perform scheduling for any arbitrary data structure composed of lists, trees, and arrays. Section 3.1 first introduces an LDS descriptor framework that captures the information required for prefetch chain scheduling. This information is also used by the prefetch engine to perform LDS traversal at runtime, described later in Section 4. Then, Section 3.2 describes a scheduling algorithm that computes the scheduling parameters (PT and PD) from the LDS descriptors.

3.1. LDS Descriptor Framework

To perform prefetch chain scheduling, the programmer or compiler first analyzes the LDS traversal code and extracts two types of information: data structure layout, and traversal code work. The data structure layout information captures the dependences between memory references, thus identifying pointer-chasing chains. Along with the data structure layout information, the traversal code work information enables our scheduling algorithm, presented in Section 3.2, to compute the scheduling parameters. In the rest of this section, we define an *LDS descriptor framework* used to specify both types of information.

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

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

In addition to nested composition, our framework also permits recursive composition. Recursively composed descriptors are identical to nested descriptors, except the dependence edge flows backwards. Since recursive composition introduces cycles into the descriptor graph, our framework requires each recursive arc to be annotated with the depth of recursion, D, to bound the size of the data structure. Figure 2d shows a simple recursive descriptor in which the inner and outer descriptors are one and the same, corresponding to a simple tree data structure. (Notice the D parameter,

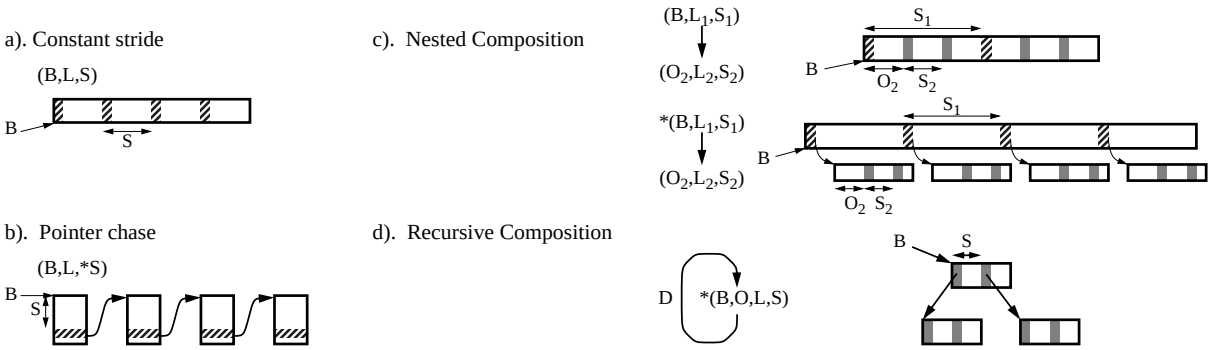

Figure 2. LDS descriptors: data layout information.

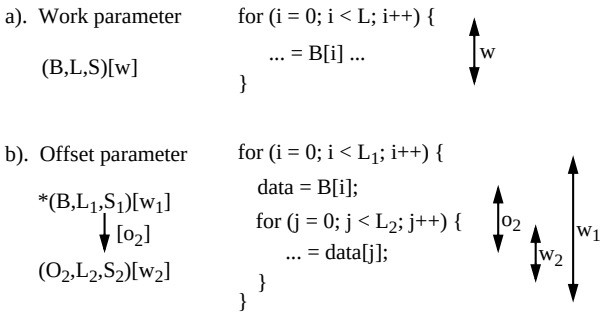

Figure 3. LDS descriptors: traversal code work information.

as well as the L parameter for linked lists, is used only for scheduling purposes. Our prefetch engine, discussed in Section 4, does not require information about the extent of dynamic data structures, and instead prefetches until it encounters a null-terminated pointer.)

Finally, Figure 3 shows the extensions to the LDS descriptors that provide the traversal code work information. The work information specifies the amount of work performed by the application code as it traverses the LDS. To provide the work information, the LDS descriptor graph is annotated with two types of parameters. The *work* parameter, w, specifies the amount of work per traversal loop iteration. Shown in Figure 3a, the work parameter annotates each array or linked list descriptor. The *offset* parameter, o, is used for composed descriptors, and specifies the amount of work separating the first iteration of the inner descriptor from each iteration of the outer descriptor. Shown in Figure 3b, the offset parameter annotates each composition dependence edge. If there are multiple control paths in a loop, we compute w and o for the shortest path. Choosing the smallest w and o tends to schedule prefetches earlier than necessary. In section 5.2, we will evaluate the effects of early prefetch arrival.

In this work we perform the code analyses to extract the descriptor information by hand. However, for basic data structures as shown in Figures 2a and 2b, descriptors can be automatically identified by a compiler [9]. For complex data structures like Figures 2c and 2d, extensions to the compiler in [9] are needed to identify the relationship between parent and child data structures.

3.2. Scheduling Algorithm

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

3.2.1. Static Descriptor Graphs

Our scheduling algorithm computes three scheduling parameters for each descriptor i in the descriptor graph: whether the descriptor requires *asynchronous* or *synchronous* prefetching, the pre-traversal time, PT_i, and the prefetch distance, PD_i. Figure 4 presents our scheduling algorithm. The algorithm is defined recursively, and processes descriptors from the leaves of the descriptor graph to the root. The "for ($N-1$ down to 0)" loop processes the descriptors in the required bottom-up order assuming we assign a number between 0 and $N-1$ in top-down order to each of the N descriptors in the graph. Our scheduling algorithm assumes there are no cycles in the graph. Our scheduling algorithm also assumes the cache miss latency to physical memory, l, is known. Due to lack of space, we are unable to describe how cyclic graphs, arising from recursively composed descriptors, are handled by our algorithm. The interested reader should refer to [8].

We now describe the computation of the three scheduling parameters for each descriptor visited in the descriptor graph. 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.[1] The "if" conditional test in Fig-

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

```
for (i = N-1 down to 0) {
    PTnest_i = max_{composed k 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 + PTnest_i                             (2)
        PD_i = ∞;                                                           (3)
    } else {
        PT_i = l + PTnest_i                                                 (4)
        PD_i = ⌈PT_i / w_i⌉                                                 (5)
    }
}
```

Figure 4. Scheduling algorithm for static descriptor graphs.

ure 4 computes whether asynchronous or synchronous prefetching is used.

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

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

3.2.2. Dynamic Descriptor Graphs

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

4. Prefetch Engine

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 3.2 to guide LDS traversal.

The hardware organization of the prefetch engine appears in Figure 5. The design requires three additions to a commodity 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 the data to the processor in 1 cycle, and also transfers the corresponding cache block from the prefetch buffer to the L1 cache.

In the rest of this section, we describe how the prefetch engine generates addresses and schedules prefetches from the LDS descriptors introduced in Section 3.1. Our 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 6a. Finally, we analyze the cost of the prefetch engine hardware.

4.1. Address Descriptor Table

The prefetch engine consists of two hardware tables, the *Address Descriptor Table* and the *Address Generator Table*, as shown in Figure 5. 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 3.2.1. 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 6b 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

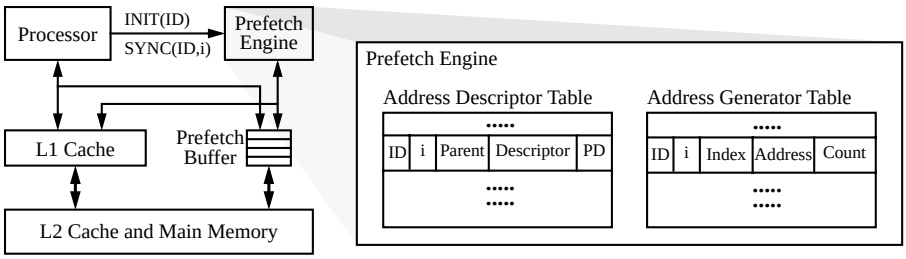

Figure 5. Prefetch engine hardware and integration with a commodity microprocessor.

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 6a does not show the ADT initialization code, our experimental results in Section 5 include all ADT initialization overheads. Notice the ADT contents should be saved and restored during context switches. Since context switches are extremely infrequent in our benchmarks, we assume the ADT contents persist across context switches and do not model the save and restore overheads.

4.2. 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 6c1 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 $(4, 0)$ 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_0, 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_0, denoted by a solid bar in Figure 6c1.

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. As shown in Figure 6c1, 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_0 to complete before it can compute its base address, $*B_0$. Eventually, the prefetch of B_0 completes, AGT entry $(4, 1)$ loads the value, and issues a prefetch for address $*B_0$, denoted by a dashed bar in Figure 6c2.

Figures 6c3 and 6c4 show the progression of the array of lists traversal. In Figure 6c3, 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 6c4. Furthermore, Figures 6c3 and 6c4 show the progress of the original AGT entry $(4, 1)$ as it traverses the first linked list serially. In Figure 6c3, 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 6c4, 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-terminated pointer is reached during traversal.

4.3. 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 6 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 or programmer to insert a $SYNC$ instruction at the top of the loop or recursive function call that traverses the data structure to provide the synchronization information, as shown in Figure 6a. 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 6c1 and 6c2. In addition, the prefetch engine "listens" for $SYNC$ instructions. When a $SYNC$ executes, it emits both an ID and an i value that matches an AGT entry. On a match, the $Count$ value in the matched AGT entry is incremented. The prefetch engine issues a prefetch as long as $Count > 0$. Once

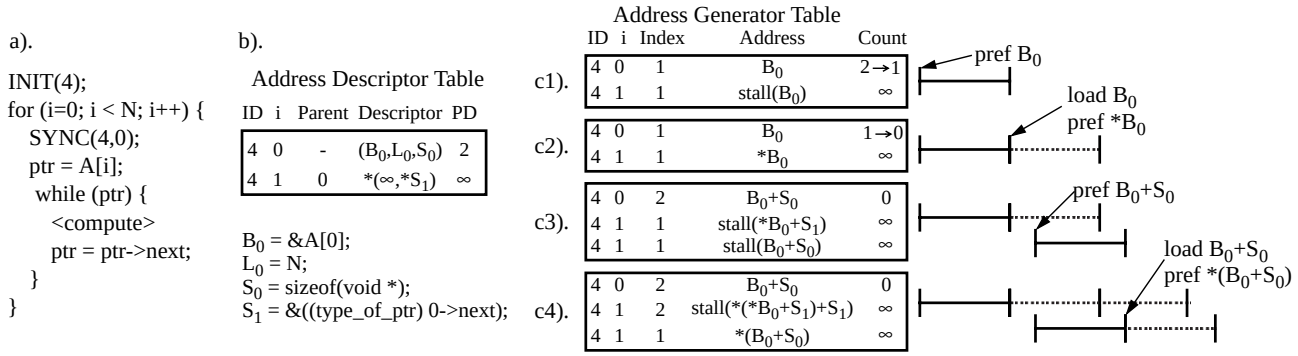

Figure 6. LDS traversal example. a). Array of lists traversal code annotated with prefetch directives. b). ADT contents. c). AGT contents at 5 different times during LDS traversal.

$Count$ reaches 0, as it has in Figure 6c2, 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 6).

4.4. Silicon Area Cost

The hardware cost of multi-chain prefetching is dictated by the sizes of the ADT, AGT, and prefetch buffer structures. The ADT should be sized to accommodate the applications' LDS descriptors. For our benchmarks, 50 entries are sufficient. Assuming each ADT entry consumes 11 bytes, the ADT size is 550 bytes. The AGT should be sized to accommodate the maximum number of active AGT entries. For our benchmarks, we found 128 entries are sufficient. Assuming each AGT entry consumes 10 bytes, the AGT size is 1280 bytes. Finally, we assume a 1-Kbyte prefetch buffer. Consequently, the prefetch engine and buffer consume roughly 2.75-Kbytes of on-chip RAM. While this analysis does not include the logic necessary to compute prefetch addresses nonetheless, we estimate the cost of multi-chain prefetching to be quite modest.

5. Results

In this section, we conduct an evaluation of multi-chain prefetching and compare it to jump pointer and prefetch array techniques. After describing our experimental methodology in Section 5.1, Section 5.2 presents the main results for multi-chain prefetching. Then, Section 5.3 examines some extensions to further improve performance.

5.1. Methodology

We constructed a detailed event-driven simulator of the prefetch engine architecture described in Section 4 coupled with a state-of-the-art RISC processor. Our simulator uses the processor model from the SimpleScalar Tool Set to model a 800MHz dynamically scheduled 4-way issue superscalar with a 32-entry instruction window and an 8-entry load-store dependency queue. We assume a split 16-Kbyte instruction/16-Kbyte data direct-mapped write-through L1 cache with a 32-byte block size. We assume a unified 512-Kbyte 4-way set-associative write-back L2 cache with a 64-byte block size. The L1 and L2 caches have 8 and 16 MSHRs, respectively, to enable significant memory concurrency. We also assume an aggressive memory sub-system with 8.5 Gbytes/sec peak bandwidth and 64 banks. Bus and bank contention are faithfully simulated.

We assume the prefetch engine has a 50-entry ADT, and a 128-entry AGT with the capability to compute an effective address in 1 cycle, as described in Section 4.4. The ADT and AGT are sized to accommodate the maximum number of required entries for our applications; hence, our simulator does not model evictions from these tables. Furthermore, we assume a 32-entry 1-Kbyte fully associative LRU prefetch buffer. Each prefetch buffer entry effectively serves as an MSHR, so the prefetch engine does not share MSHRs with the L1 cache. 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. Finally, access to the L1 cache/prefetch buffer, L2 cache, and main memory costs 1 cycle, 10 cycles, and 70 cycles respectively.

Our experimental evaluation of multi-chain prefetching uses applications from both the Olden [14] and SPECInt CPU2000 benchmark suites. Table 7 lists the applications, their input parameters and simulation windows. The columns labeled SkipInst and SimInst list the number of skipped instructions and simulated instructions, respectively, starting at the beginning of each program. The simulation windows are chosen to avoid lengthy simulations and to experiment only on the representative regions. Some applications, like EM3D, Health and MCF, have small simulation windows because their behaviors are highly repetitive. For each application, we identified the loops and LDSs to be prefetched, extracted the LDS descriptor information from Section 3.1, and computed the scheduling information as described in Section 3.2. We also inserted INIT and SYNC instructions into the traversal code. To instrument the jump pointer techniques, we followed previously published algorithms for jump pointer prefetching [9] (also known as "software full jumping" [16]) and prefetch arrays [6].

In the Olden benchmarks, there is a single "primary data structure" per application responsible for practically all the cache

Olden Benchmarks				
Application	Input Parameters	Primary Data Structure	SkipInst	SimInst
EM3D	10,000 nodes	array of pointers	27M	541K
MST	1024 nodes	list of lists	183M	29,324K
Treeadd	20 levels	balanced binary tree	143M	33,554K
Health	5 levels, 500 iters	balanced quadtree of lists	162M	505K
Perimeter	4K x 4K image	unbalanced quadtree	14M	16,993K
Bisort	250,000 numbers	balanced binary tree	377M	241,364K
SPECInt CPU2000 Benchmarks				
Application	Input Parameters		SkipInst	SimInst
MCF	inp.in		2,616M	3,200K
Twolf	ref		124M	36,648K
Parser	2.1.dict -batch < ref.in		257M	32,146K
Perl	-I./lib diffmail.pl 2 550 15 24 23 100		125M	34,446K

Figure 7. Benchmark summary.

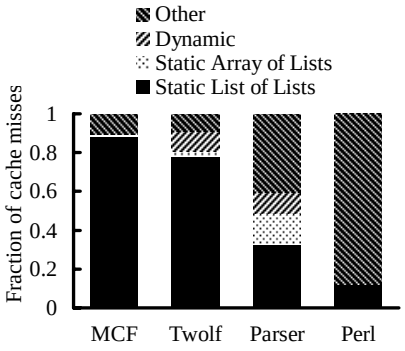

Figure 8. Fraction of cache misses suffered in static and dynamic pointer-chasing traversals in SPECInt.

misses. The SPECInt benchmarks, however, are far more complex. We used simulation-based profiling to identify the loops and data structures responsible for the cache misses in SPECInt. Figure 8 shows a breakdown of the cache misses by traversal type. Our simulations show that static traversal of list of lists and array of lists structures account for 89%, 80%, 48%, and 12% of all cache misses in MCF, Twolf, Parser, and Perl, respectively. Furthermore, dynamic traversals account for only 11% of the cache misses in Twolf and Parser. (The list of lists component for MCF is both static and dynamic, as we will explain in Section 5.3. Here, we categorize it as a static traversal.) For SPECInt, we instrument prefetching only for the loops in the static list of lists and array of lists categories reported in Figure 8. We also profiled 7 other SPECInt benchmarks, but found no measurable cache misses caused by LDS traversal.

5.2. Multi-Chain Prefetching Performance

Figure 9 presents the results of multi-chain prefetching for our applications. For each application, we report the execution time without prefetching, labeled "NP," with jump pointer prefetching, labeled "JP," and with multi-chain prefetching, labeled "MC." For 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 9 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 69% and by 40% on average for the Olden benchmarks, and by as much as 16% and by 8% on average for the SPECInt benchmarks. Comparing the MC bars versus the JP and PA bars, multi-chain prefetching outperforms jump pointer prefetching and prefetch arrays on all ten applications except MCF, reducing execution time by as much as 66% and by 28% on average for the Olden benchmarks, and by as much as 33% and by 12% on average for the SPEC benchmarks. For MCF, prefetch arrays outperforms multi-chain prefetching by 20%. In the rest of this section, we examine in detail several factors that contribute to multi-chain prefetching's performance advantage and, in some cases, disadvantage.

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. 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 62% (for MST) and 24% (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 and delete operations. In Health, jump pointer fix-up code is responsible for most of the 190% 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 EM3D 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

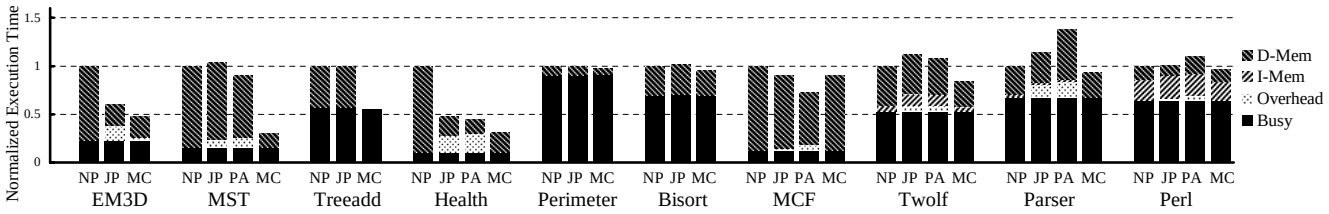

Figure 9. Execution time for no prefetching (NP), jump pointer prefetching (JP), prefetch arrays (PA), and multi-chain prefetching (MC). Each execution time bar has been broken down into useful cycles (Busy), prefetch-related cycles (Overhead), I-cache stalls (I-Mem), and D-cache stalls (D-Mem).

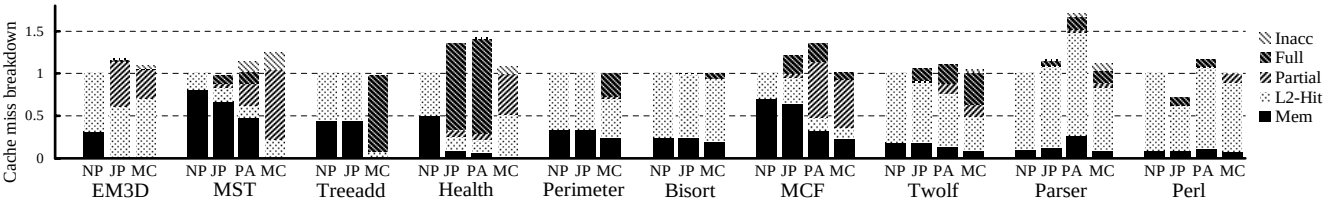

Figure 10. Cache miss breakdown. Each bar has been broken down into misses to main memory (Mem), hits in the L2 (L2-Hit), partially covered misses (Partial), fully covered misses (Full), and inaccurate prefetches (Inacc).

(at the expense of hardware support). While multi-chain prefetching requires the use of $INIT$ and $SYNC$ instructions, these instructions introduce negligible overhead. In EM3D and MCF, the traversal loops are inexpensive, hence the added code in jump pointer prefetching and prefetch arrays dilates the loop cost by 64% and 49%, respectively. Prefetch instructions also contribute to the software overheads visible in MST, Health, and the other three SPEC benchmarks. Finally, Twolf and Perl suffer increased I-cache stalls. Due to the large instruction footprints of these applications, the code expansion caused by prefetch instrumentation results in significantly higher I-cache miss rates. Since multi-chain prefetching does not require significant software instrumentation, the impact on I-cache performance is minimized.

Coverage. To further compare multi-chain prefetching and jump pointer techniques, Figure 10 shows a breakdown of cache misses. The NP bars in Figure 10 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 10 also shows inaccurate prefetches, labeled "Inacc." Inaccurate prefetches fetch data that is 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 Treeadd, Perimeter, and Bisort due to first-traversal prefetching. (First-traversal prefetching also benefits MST, but we will explain this below). 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 Treeadd and Perimeter, the LDS is traversed only once, so jump pointer prefetching does not perform any prefetching. In Bisort, the LDS is traversed twice, so prefetching is performed on only half the traversals. In contrast, multi-chain prefetching converts 90%, 28%, and 7% of the original cache misses into prefetch buffer hits for Treeadd, Perimeter, and Bisort, respectively, as shown in Figure 10. Figure 9 shows an execution time reduction of 44%, 2.4%, and 5.7% for these applications.

Figure 10 also shows the importance of prefetching early link nodes. MST and the four SPECInt 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 significantly higher cache miss coverage 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, MCF, and Twolf in Figure 9.

Early Prefetch Arrival. Figure 10 shows multi-chain prefetching is unable to achieve any fully covered misses for EM3D, MST, and Health. This limitation is due to early prefetch arrival. Because multi-chain prefetching begins prefetching a chain of pointers prior to the traversal of the chain, a large fraction of prefetches arrive before they are accessed by the processor and occupy the prefetch buffer. For applications with long pointer chains, the number of early prefetches can exceed the prefetch buffer capacity and cause thrashing. Since prefetched data is also placed in the L2 cache, the processor will normally enjoy an L2 hit where greater capacity prevents thrashing, but the L1-L2 la-

tency is exposed. This gives rise to the partially covered misses for EM3D, MST, and Health in multi-chain prefetching, as shown in Figure 10.

Despite the early prefetch arrival problem, multi-chain prefetching still outperforms jump pointer prefetching and prefetch arrays for EM3D, MST, and Health. In EM3D, limited prefetch buffer capacity causes thrashing even for jump pointer prefetching. Since multi-chain prefetching has lower software overhead, it achieves a 20% performance gain over jump pointer prefetching on EM3D, as shown in Figure 9. Comparing prefetch arrays and multi-chain prefetching for MST, we see that prefetch arrays leaves 48% 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). Consequently, Figure 9 shows a 66% performance gain for multi-chain prefetching over prefetch arrays on MST. Finally, for Health, Figure 10 shows that prefetch arrays converts 71% of the original cache misses into fully covered misses, while multi-chain prefetching converts only 49% 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 create and manage prefetch pointers in Health, multi-chain prefetching outperforms prefetch arrays by 30%, as shown in Figure 9.

Memory Overhead. In addition to software overhead for creating and managing prefetch pointers, 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 10 shows that for Health, MCF, Twolf, Parser, and Perl, the total number of cache misses incurred by prefetch arrays compared to no prefetching has increased by 41%, 36%, 11%, 67%, and 17%, respectively. (In Perl, the decrease in cache misses for "JP" is due to conflict misses that disappear after jump pointers are added, an anomaly that we did not see in any other benchmark.)

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 10 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 9 shows a net performance degradation of 39% for Parser due to prefetch arrays. For the same reasons, Twolf and Perl experience performance degradations of 8% and 10%, respectively. In contrast, multi-chain prefetching does not incur memory overhead since it does not use prefetch pointers, so the miss coverage achieved via prefetching translates to performance gains. Figure 9 shows a gain of 16%, 7%, and 1% for Twolf, Parser, and Perl, respectively.

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 adversely affect performance. In fact for MCF, prefetch arrays outperforms multi-chain prefetching, as shown in Figure 9. MCF traverses a complex structure consisting of multiple "ribbed" lists, also known as "backbone and rib" structures [16]. While each backbone and rib is traversed statically, there is very little memory parallelism. Hence, multi-chain prefetching achieves only a 9% performance gain. In contrast, prefetch arrays can still overlap misses along each backbone, and outperforms multi-chain prefetching by 20%.

5.3. Extending Multi-Chain Prefetching

In this section, we conduct a preliminary investigation of two extensions to our technique: speculative multi-chain prefetching and combining multi-chain prefetching with prefetch arrays.

As discussed in Section 2.2, compile-time analysis cannot determine the pointer-chain traversal order for dynamic traversals; however, it might be possible to speculate the pointer-chain traversal order. To investigate the potential gains of speculation, we instrumented speculative multi-chain prefetching for MCF. 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, as described in Section 5.2, 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. The "SP" bar in Figure 11 shows the execution time for MCF using speculative multi-chain prefetching. Speculation increases performance by 16% over no speculation. This leads to a performance gain of 16% over jump pointer prefetching; however, prefetch arrays still holds a performance advantage of 4%.

Another extension we evaluate is the combination of multi-chain prefetching and prefetch arrays. Multi-chain prefetching exploits inter-chain memory parallelism while prefetch arrays exploits intra-chain memory parallelism. It is natural to combine the two techniques to exploit both types of parallelism. The "SP-PA" bar in Figure 11 shows the execution time for MCF using speculative multi-chain prefetching and prefetch arrays in concert. In this version of MCF, the prefetch engine speculatively prefetches down multiple pointers at each backbone node. Then, prefetch arrays is used during the traversal of each backbone to expose intra-chain

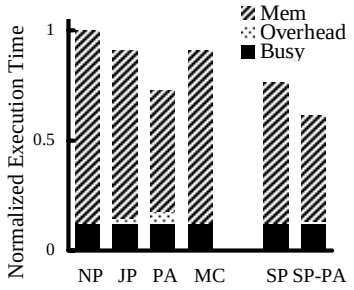

Figure 11. Preliminary performance results of speculative multi-chain prefetching and combining multi-chain prefetching with prefetch arrays for MCF.

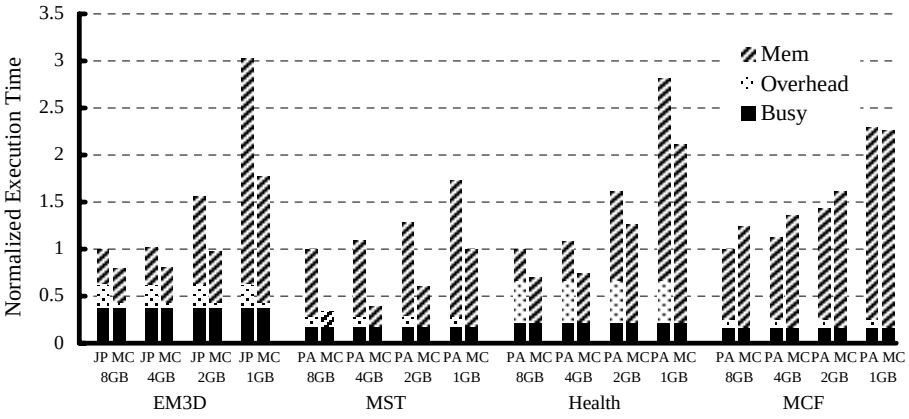

Figure 12. Prefetching performance at 8, 4, 2, and 1 GB/sec.

memory parallelism. The combined technique outperforms SP and PA in isolation by 20% and 16%, respectively.

5.4. Limited Memory Bandwidth Performance

Sections 5.2 and 5.3 examine multi-chain prefetching performance assuming an aggressive memory sub-system with 8.5 Gbytes/sec peak bandwidth. In this section, we investigate the sensitivity of our results to available memory bandwidth. Figure 12 shows the execution time for EM3D, MST, Health, and MCF as memory bandwidth is varied from 8 GB/sec down to 1 GB/sec. Results are reported for both multi-chain prefetching and prefetch arrays. The remaining applications consume very little memory bandwidth (less than 650 MB/sec average memory bandwidth), so we do not report their results.

As expected, Figure 12 shows performance degrades as memory bandwidth is decreased. However, multi-chain prefetching maintains its performance advantage over prefetch arrays for EM3D, MST, and Health at all memory bandwidths. Furthermore, the performance difference between multi-chain prefetching and prefetch arrays for MCF closes at low memory bandwidths with multi-chain prefetching slightly outperforming prefetch arrays at 1 GB/sec. Compared to multi-chain prefetching, prefetch arrays generates more memory traffic because it must fetch the prefetch pointers in addition to the application data. As memory bandwidth becomes scarce, the added traffic due to the prefetch pointers increases contention in the memory system, resulting in more memory stalls.

6. Related Work

Our work is closely related to Dependence-Based Prefetching (DBP) [15]. DBP identifies recurrent pointer-chasing loads in hardware, and then prefetches them sequentially in a prefetch engine. DBP can exploit inter-chain memory parallelism for simple backbone and rib traversals; however, DBP cannot create inter-chain overlap for more complex traversals because it pursues only one link node ahead of the CPU to minimize useless prefetches. Multi-chain prefetching uses off-line analysis to aggressively schedule inter-chain prefetches for general pointer-chasing traversals. Off-line analysis also reduces hardware complexity. However, DBP does not require programmer or compiler effort.

A follow-on paper to DBP [16] proposed Cooperative Chain Jumping, a technique that combines DBP with jump pointers for backbone and rib structures. The jump pointers create intra-chain memory parallelism along the backbone, and then DBP hardware sequentially prefetches each rib. Cooperative Chain Jumping exploits inter-chain memory parallelism; however, once again it does so only for backbone and rib traversals, and it requires jump pointers.

Unroll-and-jam [12] is a software technique that exploits memory parallelism. For applications with multiple independent pointer chains, like array of lists traversals, unroll-and-jam initiates independent instances of the inner loop from separate outer loop iterations in order to expose multiple read misses within the same instruction window. Since unroll-and-jam is a loop transformation, it must preserve the original program semantics. As a result, the scope of applicable loops are limited compared to multi-chain prefetching which annotates prefetch directives into the program code.

In addition to DBP and unroll-and-jam, three other stateless prefetching techniques have been proposed [17, 10, 9]. Also, researchers have proposed jump pointer techniques [6, 16, 9]. The merits and shortcomings of these techniques have already been discussed in Section 1, and an evaluation of multi-chain prefetching against jump pointer techniques was conducted in Section 5.

7. Conclusion

We draw several conclusions from our work. First, we conclude that inter-chain memory parallelism exists in many pointer-chasing applications. We find it is abundant in the pointer-chasing kernels from the Olden benchmark suite. Perhaps more interest-

ingly, we also find it exists in full applications from the SPECInt CPU2000 benchmark suite within several loops that perform static list of lists and array of lists traversals.

Second, we conclude that the exploitation of inter-chain memory parallelism through multi-chain prefetching provides significant performance gains. Our results show multi-chain prefetching reduces overall execution time by 40% in Olden, and by 8% in SPECInt. We also show that multi-chain prefetching outperforms jump pointer prefetching and prefetch arrays by 28% in Olden, and by 12% in SPECInt. The performance advantage of multi-chain prefetching comes from its stateless nature. Multi-chain prefetching avoids the software overheads for creating and managing prefetch pointers. In addition, multi-chain prefetching can perform first-traversal prefetching, and can effectively tolerate the cache misses of early nodes, which is important for short lists. Furthermore, multi-chain prefetching is effective for applications employing small link structures. Jump pointer techniques insert prefetch pointers that can significantly increase the number of cache misses, reducing or in some cases nullifying the performance gains derived from prefetching.

Finally, we conclude that speculation can uncover inter-chain memory parallelism for some dynamic traversals, but for MCF, prefetch arrays still outperforms speculative multi-chain prefetching by 4%. In future work, we plan to apply speculation more aggressively to see if multi-chain prefetching can outperform jump pointer techniques for dynamic traversals. However, we conclude that jump pointer techniques are preferable to multi-chain prefetching for highly dynamic traversals where inter-chain memory parallelism is difficult to identify.

We view inter-chain memory parallelism as another source of memory parallelism that supplements, rather than replaces, intra-chain memory parallelism. We believe any solution to the pointer-chasing problem should consider both forms of memory parallelism. While our results already show that the combination of multi-chain prefetching and prefetch arrays can outperform either technique alone, more work is needed to better understand when the two techniques should be combined, or when to choose one technique over the other across different applications.

8. Acknowledgments

The authors would like to thank Chau-Wen Tseng for helpful discussions on the LDS descriptor framework, and Bruce Jacob for helpful comments on previous drafts of this paper.

References

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

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

[3] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609–623, May 1995.

[4] T. cker Chiueh. Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops. In *Proceedings of Supercomputing '94*, pages 488–497. ACM, November 1994.

[5] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990. ACM.

[6] M. Karlsson, F. Dahlgren, and P. 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.

[7] A. C. Klaiber and H. 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.

[8] N. Kohout, S. Choi, and D. Yeung. Multi-Chain Prefetching: Exploiting Memory Parallelism in Pointer-Chasing Codes. UMD-SCA TR-2000-01, University of Maryland Systems and Computer Architecture Group, June 2000.

[9] C.-K. Luk and T. 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.

[10] S. Mehrotra and L. 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.

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

[12] V. S. Pai and S. Adve. Code Transformations to Improve Memory Parallelism. In *Proceedings of the International Symposium on Microarchitecture*, November 1999.

[13] S. 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.

[14] 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.

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

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

[17] C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the International Conference on Supercomputing*, May 2000.