PREFETCHING LINKED DATA STRUCTURES IN SYSTEMS WITH MERGED DRAM-LOGIC

BY

CHRISTOPHER JUSTIN HUGHES

B.A., Rice University, 1998
B.S., Rice University, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

Abstract

Recent advances in integrating logic and DRAM on the same chip potentially open up new avenues for addressing the long-standing problem of tolerating memory latency. This thesis exploits merged DRAM-logic technology to hide latency incurred by inherently serial accesses to linked data structures (LDS). We propose a programmable prefetch engine that sits close to memory and traverses LDS independently from the processor. The prefetch engine can run ahead of the processor because of its low latency, high bandwidth path to memory. This allows the prefetch engine to initiate data transfers much earlier than the processor and pipeline multiple such transfers over the network.

We evaluate the proposed memory-side prefetching scheme for the pointer-intensive Olden benchmark suite, comparing both to a system without any prefetching and to a system with a state-of-the-art processor-side software prefetching scheme for LDS. For the six benchmarks where LDS memory stall time is significant, the proposed memory-side scheme reduces execution time by an average of 27% (range of 0% to 62%) compared to a system without any prefetching. Compared to processor-side prefetching, the memory-side scheme reduces execution time in the range of 20% to 50% for three of the six applications, is about the same for two applications, and is worse by 18% for one application. We conclude that memory-side prefetching is effective, but a combination of processor-side and memory-side prefetching is best and provide a qualitative framework to determine when either scheme should be used.

Acknowledgments

I would like to thank my advisor, Sarita Adve, for her guidance without which this work would not have been possible. She gave me the freedom to pursue a problem in my own way while keeping me on track, and her enthusiasm gave me great encouragement. I would also like to thank my parents. They were a constant source of encouragement. Finally, I would like to thank my friends, both at Rice and at Illinois, who make graduate school a fun experience.

Contents

List of Tables

Table

List of Figures

Figure

1 Introduction

There has been significant progress in integrating DRAM and logic on the same chip. Several commercial products that integrate the processor and memory (and in some cases, the entire system) on a single chip are becoming available. This technology is often referred to as PIM (processor-in-memory) or IRAM (intelligent RAM) [15]. For applications whose memory requirements are met within a PIM or IRAM chip, this technology effectively addresses the long-standing memory latency and bandwidth problem. As chip sizes increase and feature sizes decrease, the application space that can exploit such chips will grow larger, and PIM chips are likely to become widespread commodity products. At the same time, there is also a large class of important applications whose memory requirements will far exceed the amount that can be put on a single PIM chip in the foreseeable future. For such applications, multiple PIM chips could be used as building blocks for larger systems. Such a system, however, reintroduces the problem of off-chip memory latency and bandwidth. Solutions that have been used to address these problems in the past for both uniprocessors and multiprocessors are applicable in the merged DRAM-logic domain as well. The ability to implement intelligent structures very close to memory, however, potentially provides new opportunities to address the memory bottleneck. This thesis explores one such opportunity.

We focus on latency hiding for linked data structures (LDS) in systems built out of multiple PIM chips. LDS are increasing in importance because of the widespread use of object-oriented programming and application domains that involve large dynamic data structures. At the same time, hiding the latency incurred in traversals of such data structures is notoriously difficult. Such traversals involve a chain of dependent loads that makes them inherently serial – the next address to be accessed is not known until the data from the previous load returns to the processor.

We propose a prefetch engine close to memory that can speculate on data that remote (or local) processors may need from that memory. In our current scheme, the speculation is aided by a software command from the processor that needs the data. Each command encodes a summary of the LDS and the path through it that is expected to be traversed. The prefetch engine uses this summary to independently perform the traversal, requesting the memory to send the traversed data to the processor. Although the prefetch engine's traversal is also serialized, its proximity to the memory results in much faster service than requests initiated at the processor. This potentially allows the prefetch engine to run ahead of the processor, initiating data transfers much earlier than the processor and pipelining multiple such transfers over the network.

The memory-side prefetching approach proposed in this thesis is in contrast to several recent

studies that have proposed prefetching LDS where all prefetches are initiated at the processor [17, 25, 26, 29, 13]. We evaluate the memory-side scheme for the Olden suite of pointer-intensive benchmarks [5]. We compare the scheme both to a system without any prefetching and to a system with a state-of-the-art processor-side software prefetching scheme for LDS based on jump pointers [17, 26]. For the six applications where LDS memory stall time is significant, the proposed memory-side scheme reduces execution time by an average of 27% (range of 0% to 62%) compared to a system without any prefetching. Compared to processor-side prefetching, the memory-side scheme reduces execution time in the range of 20% to 50% for three of the six applications, is the same for two applications, and is worse by 18% for one application. We conclude that memory-side prefetching is effective, but a combination of processor-side and memory-side prefetching is best and provide a qualitative framework to determine when either scheme should be used.

2 A Memory-Side LDS Prefetch Engine on a PIM

This section discusses our proposal for a memory-side LDS prefetch engine on a PIM.

2.1 Base Architecture

Figure 1 illustrates the architecture of a PIM node in the system we consider. Since the focus of this thesis is not to suggest an optimal hardware organization for a PIM, for simplicity, we use a common model of a multiprocessor node, but place all of the components on the same chip. This integration provides a high speed connection between the cache, memory, and the network interface. We assume that multiple PIM nodes are connected to each other via a conventional multiprocessor network in a directory-based, cache-coherent, release consistent shared-memory organization. The novel feature of our system, and our focus, is the prefetch engine on each PIM node. The engine sits next to the directory and memory, and communicates only through them.

2.2 Conveying LDS Traversal Information to the Prefetch Engine

The goal of the prefetch engine is to traverse an LDS ahead of the processor and send the accessed data to the processor before its corresponding demand access. This goal requires the engine to have knowledge of the LDS structure and the traversal path. A general way to convey this information is for the processor to download to the prefetch engine code that can be executed to traverse the LDS, and for the prefetch engine to have the ability to execute such code. In practice, we found that for most of the benchmarks we evaluated, the LDS traversal path is known *a priori*, depending primarily on the structure of the LDS. Further, this structure and traversal path can be encoded concisely in a few bytes. In cases where the exact traversal path is not known prior to the traversal, often the path depends on the results of simple comparisons involving the data within the LDS. Instructions for performing such comparisons can also be encoded in a compact manner. Therefore, in this initial study, we assume special prefetch commands that encode both the LDS structure and the traversal path, and require the programmer or compiler to insert such commands in the code before an LDS traversal. We leave the exploration of a more general determination of traversal paths in the prefetch engine to future work.

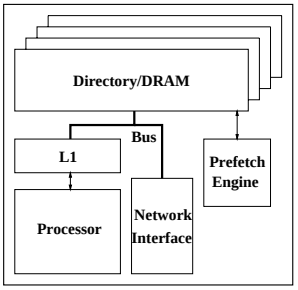


Figure 1 A single PIM with a prefetching engine

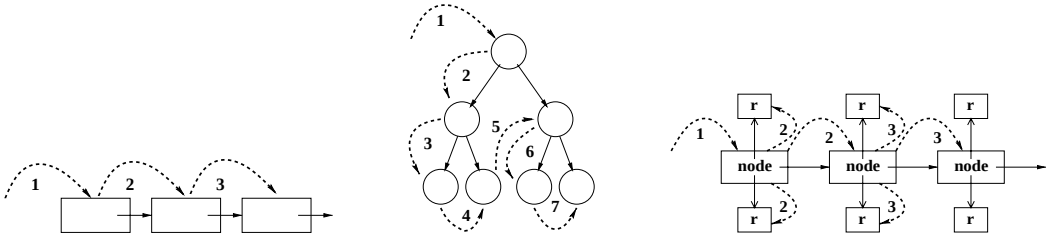
2.3 LDS Types and Traversals Supported

We support three common LDS types – linked lists, trees, and "backbone and rib" structures – as discussed below. Figure 2 illustrates supported traversals for these LDS. The dashed arrows and numbers indicate the order of traversal. The other arrows indicate pointers in the data structures.

A list traversal consists of a simple walk down the list (Figure 2(a)). In this case, the prefetch command simply needs to provide the offset into a node for the "next" pointer. The prefetch engine can then easily extract this pointer, dereference it, and continue the traversal. The command can specify that the traversal continue until a null pointer is reached or until a certain number of nodes are traversed. Some non-deterministic (input dependent) traversals are also captured. Specifically, there is support to end a traversal on satisfying a simple comparison operator on data in the traversed node and/or a specified constant (elaborated in Appendix A).

Tree traversals are more varied. In the benchmarks we examined, all deterministic traversal paths for trees are depth-first. Figure 2(b) illustrates a depth-first traversal. The prefetch engine needs to know the offsets into a node for the child pointers, which can be easily encoded in the prefetch command. The engine maintains a small hardware stack to order the nodes in the tree appropriately. This stack has to be only as large as the height of the tallest tree we wish to handle. When a node is read from memory, its child pointers are pushed on the stack and the last one is dereferenced to continue the traversal. The prefetch engine can also perform breadth-first traversals by configuring the hardware stack as a queue. As with lists, some non-deterministic traversals can be specified by using simple comparison operators on node data and/or constants to determine which successor pointers to follow.

Backbone-and-ribs structures are LDS with one or more data pointers inside each node [26]. The LDS is the "backbone" and the data to which the nodes point are "ribs" (Figure 2(c)). Although



(a)A linked list traversal (b)A DFS tree traversal (c)A "backbone-and-ribs" traversal – the nodes with r's are the ribs

Figure 2 Supported LDS and traversal types. The dashed arrows with numbers indicate the order of traversal. The other arrows are pointers in the data structures.

the ribs are not an LDS, their traversal is dependent upon the traversal of the backbone. Therefore, prefetching the ribs in parallel with the backbone LDS would be beneficial. For the prefetch command, simple information about the offsets into a node for the "next" backbone pointer and for the ribs can be specified analogous to lists.

We implemented a prefetch command that encodes the above options in two words. Appendix A describes the prefetch command format. Our prefetch command could also be used to prefetch arrays. We did not use the prefetch commands for arrays in our benchmarks since arrays reads are not the primary cause of memory stall time for them.

2.4 Executing a Traversal

A processor issuing a prefetch command sends it to the directory of the node that contains the first element of the LDS to be traversed. The directory forwards the command to its prefetch engine. The prefetch engine issues a prefetch to the directory for the first address associated with the request. The prefetch is given lower priority than demand requests, but is otherwise treated like a demand until the data is retrieved from the DRAM. Then, in addition to sending a prefetch reply to the requesting processor, a copy of the data is sent to the prefetch engine. When the last line for a node returns, the prefetch engine translates any successor or rib pointers from virtual to physical addresses and places them on the stack (of which simple list traversals will only use a single entry). If required, comparisons are first performed to determine the correct successor pointer to follow. The address translation is fast because the prefetch engine has access to the TLB on its PIM chip, as discussed further in Section 2.8. The engine then issues a prefetch for the next node by grabbing its address from the stack. Any ribs associated with the last node prefetched can also be prefetched at this time (in the same manner), concurrently with the next node. When the traversal is complete

(as determined from the prefetch command), the prefetch engine requests the directory to send a reply to the requesting processor to inform it of the completion.

2.5 Command Migration – Dealing with Data Distributed on Multiple PIMs

The entire LDS to be traversed is not guaranteed to be present in a single PIM's memory. It is possible that the nodes in an LDS are scattered throughout the system, making it critical to have the ability to run the prefetch command on different PIMs. For a list, this is relatively straightforward. If the next node in a list is on a different PIM, then we can *migrate* the prefetch request to that PIM. The requesting processor does not have to know that its request has been moved.

Migrating a request for a tree traversal is more complex. Recall that a tree traversal requires some state information to order the nodes. There are multiple possible policies regarding when a command with state should migrate. Tree traversals requiring migrations are not critical for any of the applications studied. Therefore, we implemented a simple policy. We migrate requests only when there is no remaining state; otherwise, the request for a cache line on another PIM is discarded.

2.6 Cache-Coherence Issues

Memory-initiated communication can introduce new race conditions in a cache-coherence protocol. To simplify a number of cases, we pursued only read prefetching. Many LDS traversals in the benchmarks are read-only. Further, nodes that are modified are rarely being shared at the time of modification. We use a MESI cache coherence protocol; therefore, such nodes are returned in exclusive state even with a read prefetch. This allows the processor to write to such nodes without sending out upgrade requests.

The key new race condition introduced is when a directory receives a request for data that it believes is already at the requesting processor. This situation is not a new case for coherence hardware, per se, but rather it can be caused by some new conditions that require different handling. The two new conditions that can cause it are a prefetch arriving for data that has already been read by another prefetch or demand access by the same processor, and a demand access arriving for data that has just been prefetched by the same processor.

In the first case, the prefetch can simply be squashed. The prefetch engine still needs access to the data in order to continue the traversal, but the data is not returned to the processor. Since the

processor does not notify the directory on eviction of a shared line, this could lead to unnecessary squashing of prefetches and reduced prefetch coverage. However, we observed that the increase in network traffic due to sending the unnecessary data outweighs the benefits of higher coverage. This is true for the benchmarks examined primarily because much of the LDS data is not reused.

In the second case (i.e., the directory receives a demand request for a line it believes to already be in the requesting processor's cache in an acceptable coherence state), there are two options. The request can wait at the directory in case it overtook a writeback from its processor for that line, or it can go back to its origin and determine if it is still necessary. It is possible that a prefetch for the requested line occurred too late to prevent the demand request from going out and so the demand is unnecessary. In our system, we send all such demands back to the cache to check if they were unnecessary. This technique will increase the response time for demands that overtake writebacks, but that is typically an infrequent occurrence.

2.7 Coalescing Demand Requests with Prefetches

Traditional prefetches (to individual cache lines [20]) that are sent to memory occupy miss status handling registers (MSHRs), or an equivalent resource, in the processor's cache. If a traditional prefetch is late (returns after the processor requests the data), then any demand access to the same line by the processor will coalesce with the prefetch in the MSHR. This prevents redundant demands from being sent to memory and enables the prefetch to overlap part of the latency that would have been seen by the demand. In our system as described thus far, if a prefetch is late, a redundant demand will be sent out. This is because the cache does not know which lines are being prefetched. We add the following hardware to the cache to allow it to predict which lines are being prefetched so that it can avoid sending redundant demands to the directory.

We expand the MSHRs to have space for holding information about prefetch commands. We reserve an MSHR for each outgoing prefetch command and we use a unique identifier to match prefetch responses with MSHRs when they return. We place address generation hardware analogous to the prefetching engine's hardware next to the MSHRs. The hardware uses the node data returned by the prefetching engine to predict which line is going to be sent next by the prefetch engine. This prediction is used to coalesce a subsequent demand request for the predicted line with the prefetch command in the MSHR. If a demand request coalesces with a prefetch command and the next line returned is not the same as the predicted one or the traversal is completed (for reasons explained below), then the cache recognizes that it mispredicted. In that case, it sends the coalesced demand

request down, albeit delayed.

We are not guaranteed that the above hardware will predict correctly because the prefetch engine will not return a line if the directory's state shows it as already being in the processor's cache (as discussed in Section 2.6). Since the hardware at the cache does not know with which lines this will occur, it conservatively assumes that all of the nodes will be returned. A possible way to improve prediction accuracy is for the hardware to lookup a node in the cache to see if it is already present. If so, the prefetch engine definitely will not return that node. However, this would create more contention for the cache ports. We henceforth refer to the above hardware as the *prefetch predictor.*[1]

2.8 Support for Address Translation

The prefetch engine and the address generation hardware at the cache dereference pointers, which requires a virtual to physical address translation. We propose to use the processor data TLB on the same PIM for this purpose. The processor is always given priority when accessing the DTLB. Our implementation assumes a hardware DTLB miss handler similar to that in Intel's Pentium family of processors [10]. We do not increase the size or the number of ports on the DTLB for our scheme. Thus, its use by the prefetch hardware could increase contention and misses; both effects are modeled in our simulations and their impact is discussed in Section 4.1.3.

2.9 Support to Mitigate Effects of Early and Useless Prefetches

The prefetch engine could potentially hurt performance by causing cache pollution in at least two ways. First, the prefetch engine's traversal may get too far ahead of the processor. In that case, the prefetched data could arrive too early and replace a useful line that will be accessed before the prefetched data. Second, for some traversal paths that cannot be captured exactly by our prefetch command, the prefetch engine may continue to traverse the LDS even after the processor has terminated its traversal or may traverse incorrect paths. In this case, the prefetch engine may cause cache pollution by sending useless data to the processor. One method to avoid such pollution is to support a prefetch buffer (that is exposed to the cache-coherence protocol), and deposit prefetched

[1] An alternate scheme to avoid using prediction hardware is for the cache to interpret a prefetch response as a response to any outstanding demand request to the same line. The reply to the demand request would need to be discarded. No prediction would be used, which could increase the ratio of useful prefetch transfers. However, a useless demand request would be generated for each late prefetch. This creates extra contention in the network and at the DRAM. We found that this contention overcomes the benefits of this scheme.

data into this buffer rather than the cache. For the case where the processor terminates the LDS traversal before the prefetch engine, a stop command could be sent to the prefetch engine. We did not include support for any such hardware to handle early or useless prefetches, and justify this decision in Section 4.1.3.

3 Experimental Methodology

3.1 Evaluation Environment and Architecture Modeled

We modified the RSIM simulator [23] to model the base architecture with and without memory-side prefetching, as discussed in Section 2. The processor-side schemes we simulated do not require any hardware support except for a conventional prefetch instruction, and are discussed further in Section 4.2. RSIM models a state-of-the-art superscalar out-of-order processor, memory system, and network in detail, including contention at all resources.

Table 1 summarizes the system parameters for the base architecture simulated. Since the instruction footprint of our benchmarks is small, we assume that all instructions hit in the instruction cache and in the instruction TLB (with a single cycle hit time). The data cache size chosen is sufficient to hold the first-level working sets for all benchmarks, but not large enough to hold their second-level working sets. This follows the methodology of Woo et al. [27] which suggests scaling down the data cache sizes based on application input sizes (which are typically scaled down for simulation).

Since main memory latency is greatly reduced on a PIM, the gap between main memory latency and second-level cache latency is narrowed considerably. This reduces the performance benefits seen from a second-level cache on a PIM, and may make such a large structure no longer cost-effective. Therefore, our simulated base architecture reported here contains only a single level of cache. We, however, also performed our experiments on an identical architecture with a second-level cache. The results from these were similar to those with a single-level cache.[2]

To determine the sensitivity of our results to memory latency, we also performed experiments with a processor that is twice as fast as the processor in the base architecture, keeping all the latencies in the memory hierarchy the same (in ns). The results were qualitatively the same as those with the base latencies.[3]

[2]Processor-side prefetching is capable of hiding second-level cache hit latency. Our memory-side scheme cannot since it only prefetches data from main memory. Therefore, it was possible that the processor-side scheme could have seen a significant improvement for a system with a second-level cache. However, the benchmarks used have little data reuse, and the out-of-order processors are able to hide much of the second-level cache latency.

[3]Quantitatively, we found that for all benchmarks except for *treeadd*, there is very little change ($< 6\%$) in the relative performance difference among all systems evaluated. For *treeadd*, which is the only benchmark where processor-side prefetching significantly outperforms memory-side prefetching, the difference in the two schemes widened further (from 18% to 36%).

Memory Hierarchy and Network Parameters		ILP Processor	
L1 cache (on-chip)	64K, 2-way associative, 64B line, 2 ports, 8 MSHRs	Processor Speed	600MHz
		Fetch/Retire Rate	4 per cycle
Bus (on-chip)	600 MHz, 128 bits, split trans.	Functional Units	2 Int, 2 FP, 2 Add. gen.
Memory (on-chip)	4-way Interleaved, 30ns access, 16B/cycle	FU Latencies	1/3/9 int. add/mult./div. 3/4/10 FP add/mult./div.
Network	2D mesh, 64 bits, 4 cycle flit delay per hop	Instruction window (reorder buffer) size	64 entries
PIMs in system	16	Memory queue size	32 entries
DTLB	128 entries, fully associative, hardware-managed, 2 ports, 30 cycle miss penalty	**Contentionless Memory Latencies**	
		L1 hit time (on-chip)	1 cycle
		Local Memory (on-chip)	26 cycles
		Remote Memory	90-170 cycles
		Cache-to-cache transfer	108-201 cycles

Table 1 Parameters for the base system

3.2 Evaluation Workload

To evaluate memory-side prefetching, we use the Olden benchmark suite, which is a collection of pointer-intensive codes [5]. We present results for seven of the ten Olden benchmarks; the other three either have very small memory stall time ($< 10\%$ for *barnes* and *power*) or are dominated by dereferencing of computed addresses rather than pointers (in *voronoi*). Table 2 provides a brief description of the benchmarks studied and the input parameters used.

The Olden suite was written to be compiled by an automatically parallelizing compiler. The benchmark source codes include hints to the compiler on how they can be parallelized. We manually parallelized four of them, *em3d*, *health*, *perimeter*, and *treeadd*, faithfully following the parallelization technique suggested to the compiler. If possible, LDS nodes are placed on the PIM that is most likely to traverse them. (This may not always be possible since in the simulated system, data placement is done at the granularity of a page.) The other three benchmarks, *bisort*, *mst*, and *tsp*, do not scale well on a multiprocessor, so we chose to leave them as uniprocessor programs but examine their

Benchmark	Description	LDS prefetched	Input data size
bisort	Performs ascending and descending bitonic sorts	Dynamic binary tree	64K nodes
em3d	Simulates propagation of EM waves in a 3D body	Static linked list with ribs	4K H nodes 4K E nodes
health	Simulation of the Columbian health care system	Dynamic linked lists	level=5 time=300
mst	Builds a minimum spanning tree	Static linked lists	1024 nodes
perimeter	Computes the perimeter of regions in images	Static four-way tree	1K x 1K image
treeadd	Sums the values in a tree	Static binary tree	1M nodes
tsp	Traveling salesman problem	Dynamic linked lists	64K nodes

Table 2 Benchmark characteristics

behavior when their data is (randomly) distributed amongst multiple PIMs at page granularity (this would be the case if the data set were too large to fit in a single PIM's memory).

The prefetch commands for memory-side prefetching were inserted by hand. In general, we attempted to minimize the number of useless prefetches issued. At times, this forced us to place the instructions very close to the first access to the LDS. For the benchmarks studied, *health*, *mst*, and *tsp* use list traversal prefetch commands, *bisort*, *perimeter*, and *treeadd* use tree traversal commands, and *em3d* uses a list traversal with ribs command. Both *mst* and *tsp* issue commands that use the prefetch engine's comparison hardware. In *mst*, the engine performs traversals of hash table buckets, where it stops the traversal if a node with the matching key is found. For *tsp*, the engine traverses circularly linked lists, and it stops once the first node in the list is reached again.

For all of the benchmarks except *bisort* and *perimeter*, we attempt to prefetch the primary LDS every time it is accessed. These benchmarks make a single pass through the LDS, or the passes are far enough apart (for *health*) or the LDS is large enough (for *tsp*) that most of the nodes are not in the cache when we begin prefetching again. *Bisort* and *perimeter* make multiple passes through their primary LDS, and they traverse only part of the LDS each time. Their LDS traversals cannot be exactly captured by our prefetch command. *Bisort* uses comparisons involving data from multiple nodes to determine traversal paths. *Perimeter* traverses up and down an unbalanced binary tree, where the traversal path is dependent upon the shape of the tree. While it is possible to attempt to prefetch along all possible paths for these two benchmarks (as in [13]), bandwidth can become a limitation and degrade performance. Therefore, we issue prefetches for only a very small number of the LDS node accesses. The methodology used for processor-side prefetching is discussed in Section 4.2.3.

All benchmarks except *health* contain large initialization phases where the data structures are built (*health* builds only a small structure on initialization). We only report results for the computation phase of each benchmark since this phase would be repeated many times in realistic executions of the applications, making the initialization a negligible part of the overall execution time. For *health*, the primary linked data structures are linked lists that start off empty. Therefore, we begin recording results after 250 iterations in the computation phase have passed, to allow the benchmark time to grow the lists to a more "steady state" configuration.

The multiprocessor benchmarks (*em3d*, *health*, *perimeter*, and *treeadd*) are run on sixteen processors. The uniprocessor benchmarks (*bisort*, *mst*, and *tsp*) are run on one processor, but have the nodes in their critical LDS randomly allocated across sixteen PIMs.

3.3 Evaluation Metrics

The primary metric used in our evaluation is execution time. For further insight, execution time is divided into six components – busy time, functional unit stall time, local memory stall time (stall time for memory accesses resolved within the local PIM, either at the L1 cache or local DRAM), remote memory stall time (stall time due to remote memory accesses and cache-to-cache transfers), TLB miss stall time (stall time waiting for the TLB miss handler to complete), and synchronization stall time. The busy and stall times are calculated as follows, similar to previous work [22]. For each cycle, we calculate the ratio of instructions that are retired to the maximum retire rate and record this as busy time. The remaining fraction of that cycle is charged as stall time for the first instruction in the instruction window unable to retire; however, if the first instruction is waiting for a TLB miss to be resolved, the time is charged as TLB stall time. We also collect a variety of statistics about prefetches, as explained in Section 4.

4 Results

This section presents our results. For each benchmark, Figure 3 shows the normalized execution times for the base system without prefetching (Base), the base system with memory-side prefetching (MPF), and the base system with processor-side prefetching (PPF). The bars are normalized to the time for Base for the corresponding benchmark. Each bar is split into the busy and stall components of execution time as described in Section 3.3. Section 4.1 discusses the impact of memory-side prefetching on the base architecture, focusing on the Base and MPF bars in Figure 3. Section 4.2 describes the processor-side prefetching schemes evaluated and compares memory-side prefetching with processor-side prefetching, focusing on the MPF and PPF bars in Figure 3.

4.1 Impact of Memory-Side Prefetching on the Base Architecture

4.1.1 Overall Results

Figure 3 shows that five of the benchmarks incur significant improvements in performance from memory-side prefetching (19% to 62%), while the other two, *bisort* and *perimeter*, see an insignificant change. The average reduction in execution time over the six benchmarks with significant LDS stall time (i.e., excluding *perimeter*) is 27%.

As discussed in Section 3.2, the five benchmarks where we see significant improvements have traversal paths that can be captured by the implemented prefetch command (Section 2.3 and Appendix A). To effectively prefetch LDS for applications with more complex traversal paths (such as *bisort* and *perimeter*), more general-purpose prefetch engine hardware is required as discussed in Section 2.2. We leave the exploration of such hardware to future work.

The next few sections analyze the above results in more detail, discussing the causes for the benefits and limitations of the memory-side prefetching scheme for the benchmarks where our prefetch command was applicable. We do not discuss *bisort* and *perimeter* any more since very few prefetches were used for these benchmarks and they did not have a significant impact.

4.1.2 Understanding the Benefits and Limitations of Memory-Side Prefetching

To understand the benefits and limitations of memory-side prefetching, we examine prefetch coverage (Table 3) and the fraction of prefetch data transfers (as opposed to aggregate prefetch commands) that are useful, late, and damaging (Table 4(a)). (These terms are explained below.) Additionally,

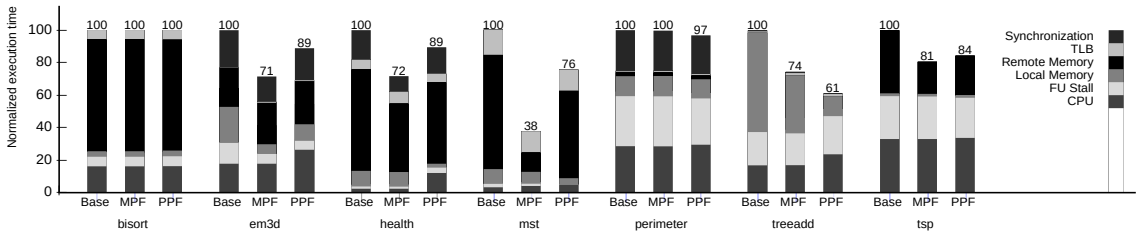


Figure 3 Normalized execution times. Results for base system (Base), memory-side prefetching (MPF), and processor-side (jump-pointer) prefetching (PPF).

Table 4(b) provides supplemental data to explain some of the above prefetch statistics. Finally, we also discuss the impact on bandwidth.

Prefetch Coverage

We measure prefetch coverage as the reduction in demand read miss requests. Although we only prefetch LDS accesses, the prefetch coverage measures the read misses reduced as a fraction of all (including non-LDS) read misses. For the prefetching scheme to be effective, it must have high prefetch coverage. Table 3 gives the reduction in demand read miss requests, as well as a breakdown of these requests into local and remote reads. We see that all five benchmarks have large reductions in the number of demand read misses, indicating high prefetch coverage (over 50% for all of the benchmarks). Further, the prefetch engine is able to reduce demand reads from both local and remote memories; its behavior is not specialized to a specific memory latency. The reason that the coverage is not higher for these benchmarks is that there are a significant number of non-LDS read misses. (They could potentially be prefetched with more conventional prefetching [20].) Also, for *health* and *treeadd*, sometimes the prefetch predictor mispredicts, making some prefetches useless and reducing the coverage, as elaborated next.

Useful and Useless Prefetch Data Transfers

A prefetch data transfer (as opposed to an aggregate prefetch command) is useful if the data it returns is used by the processor before being evicted. However, if the prefetch predictor at the cache mispredicts and sends a demand request for a line for which a prefetch is already on its way back from memory, then the corresponding prefetch data transfer is not counted as useful. Prefetch transfers that are not useful (called useless prefetches) are wasteful of resource bandwidth, and so can reduce performance.

Table 4(a) shows that the fraction of prefetch data transfers that are useful is large for all five benchmarks, and over 95% for *em3d*, *mst*, and *tsp*. The reason for the relatively lower number of

Benchmark	Prefetch coverage (% Reduction in demand read misses)	Local		Remote	
		% Base	% Reduction	% Base	% Reduction
em3d	84.3	94.5	88.5	5.5	11.9
health	53.5	37.4	15.0	62.6	76.5
mst	57.0	45.9	7.4	54.1	99.0
treeadd	67.9	100.0	67.9	0.0	N/A
tsp	70.1	7.5	68.7	92.5	70.2

Table 3 Prefetch coverage

Benchmark	% Useful prefetch transfers	% Late prefetch transfers	% Damaging prefetch transfers
em3d	96.9	4.9	2.1
health	89.1	71.0	4.3
mst	99.9	61.5	0.0
treeadd	73.5	0.2	0.2
tsp	97.6	96.2	1.6

(a)

Benchmark	% Missed Coalesce	LDS Nodes Traversed/ Command	Migrations/ LDS Node Traversed
em3d	1.0	256.0	0.0
health	8.5	57.4	0.89
mst	0.0	3.0	0.96
treeadd	26.5	65535.0	0.0
tsp	0.2	668.0	0.11

(b)

Table 4 Prefetching statistics. (a)Useful, late, and damaging prefetches. (b)Supplemental data to explain useful, late, and damaging prefetches.

useful prefetches for *health* and *treeadd* is explained by the data in the first column of Table 4(b). This column contains the fraction of prefetch data transfers that could have had a corresponding demand request coalesce with their prefetch command, but did not because of a misprediction by the prefetching predictor. In these cases the data transferred by the prefetch is not used. The *health* and *treeadd* benchmarks have a significant number of missed coalesce opportunities, explaining the lower fraction of useful prefetch transfers (and relatively lower prefetch coverage). For *health*, mispredictions are caused by data reuse. For *treeadd*, they arise because the prefetch engine runs too slowly. The processor can potentially send out multiple requests in parallel when traversing leaves of the tree. If the prefetch engine keeps ahead of the processor then this will not happen. However, the prefetch engine *is* too slow (for reasons explained below), and the processor does send out multiple LDS node requests concurrently. Since the prefetch predictor assumes a completely serial traversal, this creates mispredictions.

Late Prefetch Data Transfers

A late prefetch data transfer is a useful prefetch, but one that does not arrive early enough to avoid exposing memory stall time. A late prefetch is unable to hide the entire memory latency, but still improves performance by hiding some latency. Note that late prefetches always have a demand read coalesced with them, and would not occur if the cache did not have prediction hardware to allow coalescing with prefetch commands. Without such hardware, a prefetch considered late here would be useless since the corresponding demand miss would always be sent out for the data no

matter how close the prefetch is to arriving at the cache.

Table 4(a) shows a significant number of late prefetch data transfers for *health*, *mst*, and *tsp*. These occur due to two factors, as explained below.

First, a late prefetch data transfer can occur if there is too little computation per LDS node and the corresponding prefetch command is not issued early enough (relative to the length of the LDS). The prefetch engine traverses the LDS one node at a time, serialized by the DRAM access time. If this serialization time is larger than the computation time per LDS node at the processor, then the processor could run ahead of the prefetch engine and incur late prefetches. This effect could be mitigated, however, if the prefetch command is sent out well before the processor begins its LDS traversal and the LDS is short enough that most prefetched data arrives before the corresponding demand access. *Health* and *mst* have small computation time per node, but are able to send prefetch commands early and have short LDS as seen from the second column of Table 4(b) (more so for *mst*). *Tsp* has a moderate amount of work per node, but not quite enough to cover the memory latency. It is unable to send out prefetch commands early enough for its long LDS, and therefore sees late prefetches.

Another cause for late prefetches is prefetch command migrations (third column of Table 4(b)). A migration delays the LDS traversal process, increasing the chance of a prefetch transfer being late. All three of the aforementioned benchmarks have significant migration rates, especially *health* and *mst*.

Damaging Prefetch Data Transfers

A prefetch data transfer is damaging if it replaces a line that is needed by a subsequent demand access. Overall, the number of damaging prefetches is very low for all five benchmarks (less than 5%).

The number of damaging prefetches is largely dependent on the number of prefetch transfers that return too early or that are not useful. The early prefetches result from two factors – the placement of the prefetch commands in the code and the amount of work done by the processor per node. First, if a prefetch command is issued well before any use of the LDS data, then prefetch transfers may return too early and cause the eviction of useful data. This is the cause of the 4% damaging prefetch data transfers in *health*. There is a balance in *health* between prefetching some data too early and placing the prefetch command too late; the implementation used provided optimal performance. Second, if a processor performs a lot of work on some nodes, then the prefetch engine may get too far ahead of it and return some data too early. This is the case in *em3d* and *tsp*. (*Mst* and *treeadd*

have negligible damaging prefetches.) Nevertheless, in all cases, the number of damaging prefetches is low enough to not have any significant impact on performance.

Network, memory, and directory bandwidth

The memory-side prefetching scheme issues a single prefetch command for multiple LDS node requests, potentially reducing the network bandwidth requirement. However, missed coalescing opportunities and useless prefetches can increase network, memory, and directory bandwidth requirements, and migrations of unnecessary prefetches can increase the network bandwidth requirement. For the architecture and benchmarks studied here, however, bandwidth at these resources did not turn out to be a performance limitation (e.g., request network utilization is under 10% for all cases except for *health* with prefetching; in the latter case it was 14% due to migrations.

Summary

In summary, our prefetch command is able to capture most LDS traversals in our benchmark suite. For the benchmarks with such traversals, the memory-side prefetching scheme is effective at reducing both local and remote memory stall time. The prefetch coverage is fairly high, and the number of useless prefetches issued is relatively small since there is little speculation involved. The performance benefits are primarily limited by the DRAM latency and migration rate relative to the work done per node. This is manifested as late prefetches for *health*, *mst*, and *tsp*. For *treeadd*, this is instead manifested as useless prefetches. Note that in *em3d* and *health* synchronization stall time is also reduced, due to a reduction in load imbalance. In all benchmarks, some of the remaining latency is due to non-LDS accesses, as they were not addressed in this paper. Finally, damaging or early prefetches were not a limiting factor in our benchmarks.

We note that memory-side prefetching is beneficial even for the uniprocessor benchmarks, *mst* and *tsp*. The system considered in this paper has a processor in every memory chip. However, these results show that merged-DRAM logic chips with only the proposed prefetch engine in place of conventional DRAM can also exploit the benefits of memory-side prefetching.

4.1.3 Evaluation of Features of the Prefetching Hardware

This section examines some of the specific design decisions made for the prefetching hardware.

Support for prefetch command migration. Table 4(b) shows that *health*, *mst*, and *tsp* have a relatively large number of prefetch command migrations. This is because the traversals in these benchmarks are of linked lists with nodes scattered throughout the system. These migrations slow the LDS traversal and incur late prefetches; however, without the migrations the traversal would

terminate, reducing prefetch coverage and the benefits from prefetching. The larger the LDS and the larger the migration rate, the more important migrations become for high prefetch coverage. For *mst*, only 33% of the nodes would be prefetched without migration, but the number drops to 2% for *health* and is even smaller for *tsp*. Migration can also be important for applications with very low migration rates; in such cases many subsequent nodes may otherwise be left untraversed by the prefetch engine. This is the case for *em3d*, where almost 50% of the performance benefit is lost without migrations. Thus, support for migration of prefetch commands is critical.

Support for coalescing demand misses with prefetch commands. Another critical feature of the prefetching hardware is the ability to coalesce demands with prefetch commands. The importance of this feature is illustrated by the high fraction of late prefetch data transfers for many of the benchmarks (Table 4(a)). Although the late prefetch transfers are able to hide only part of the memory latency, they would be useless if a demand was not coalesced with them (and potentially reduce performance by unnecessarily increasing traffic).

Lack of support for early and useless prefetches. The fraction of damaging prefetches is quite small for all benchmarks (Table 4(a)), showing that, in general, there is little cache pollution due to early or useless prefetches. This shows that our decision to not include hardware to handle such a case (as described in Section 2.9) was justified, at least for the benchmarks studied. It is possible, however, that for other applications, such hardware may be needed.

DTLB hardware. The results in Figure 3 show that our decision to use the processor data TLB for the prefetch engine (without any increase in its size or number of ports) was justified. All benchmarks other than *bisort*, *health*, and *mst* have an insignificant DTLB stall time. Our prefetching scheme does not impact *bisort*. For *health*, the DTLB miss rate increases, but the resulting impact on execution time is small. For *mst*, there is no increase in DTLB miss rate or stall time. None of the benchmarks sees any significant DTLB contention.

4.2 Comparison of Memory-Side and Processor-Side LDS Prefetching

This section compares the effectiveness of memory-side LDS prefetching proposed in this paper to processor-side LDS prefetching proposed in previous work. The most promising processor-side LDS prefetching techniques use jump-pointers [17, 26]. While these techniques were developed for uniprocessor systems, they can be applied to multiprocessors as well. We use software jump-pointer prefetching techniques as proposed by Roth and Sohi [26] to represent processor-side prefetching for the bulk of our analysis. Section 4.2.1 provides background on these techniques. Section 4.2.2

provides a qualitative analysis of when they can be expected to be outperformed by memory-side prefetching and vice versa, and Section 4.2.3 presents quantitative results. Section 4.2.4 analyzes alternative jump-pointer prefetching implementations. Section 4.2.5 provides a comparison with another processor-side technique called prefetch arrays [13].

4.2.1 Background on Jump-Pointer Prefetching

Jump-pointer prefetching augments each data structure with additional pointers that are not originally present in the application. These *jump-pointers* are set to point to LDS nodes that will be accessed multiple iterations or recursive calls in the future. When an LDS node is visited, prefetches are issued for the locations pointed by the jump-pointers in the node. Prefetching nodes multiple iterations or recursive calls before their demand use enables their access latency to be hidden. Creating and updating the jump-pointers, however, can involve significant overhead for applications using dynamic structures. Dynamic structures are those that are updated during the computation phase. Static structures incur the overhead only once, and generally in the initialization phase of the application.

Roth and Sohi propose four different idioms for jump-pointer prefetching – queue jumping, full jumping, chain jumping, and root jumping [26], as described below.

Queue jumping. This is the simplest idiom of jump-pointer prefetching. Each LDS node has a jump-pointer that points to another node in the same LDS that is likely to be accessed in the near future (Figure 4(a)). Just before beginning work on a node, a prefetch is issued for the node pointed by its jump-pointer. The distance between a node and the node pointed by its jump-pointer is called the *jump interval*. As long as the jump interval is large enough, the prefetch will complete before its corresponding demand access. Further, multiple jump-pointers from different nodes may be outstanding at a time, achieving some parallelism in the memory system. The first few nodes of an LDS, however, have no jump-pointers pointing to them, and so are not prefetched. The processor will access these nodes sequentially, and will most likely stall waiting for the loads for these nodes to complete. We call this the *startup* period. During the startup period, jump-pointers for later nodes are prefetched. The startup period ends after one jump interval, leading to a *steady state* in which prefetches are expected to complete before the corresponding demand accesses. For small LDS (relative to the jump interval), the startup period can dominate, reducing the potential benefits from queue jumping.

Full jumping. This is a variant of queue jumping for backbone-and-rib structures. Here each node

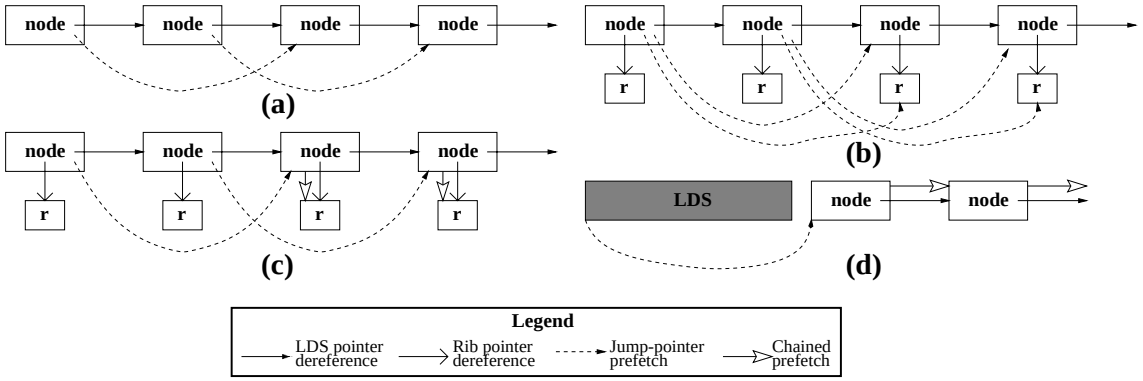


Figure 4 The four jump-pointer prefetching idioms. (a) Queue jumping. (b) Full jumping. (c) Chain jumping. (d) Root jumping.

has additional jump-pointers to rib structures of another node (Figure 4(b)). This technique allows rib structures to be prefetched in parallel to the backbone of an LDS, and has the same potential benefits and limitations as queue jumping.

Chain jumping. A different method of prefetching rib structures that does not require jump-pointers for ribs is chain jumping. The backbone of the LDS is prefetched using queue jumping, but the ribs are prefetched using built-in (natural) pointers of the original data structure (Figure 4(c)). The rib prefetches are called chained prefetches. Generally, the processor issues chained prefetches from a node in the same iteration that it issues the jump-pointer prefetch for the node. In that case, if the jump-pointer prefetch does not complete by the end of the iteration, the processor will stall because of the address dependency for the chained prefetch. A solution to this is to double the jump interval and stagger the jump-pointer and chained prefetches for a particular node and its ribs. With the same technique used to create jump-pointers, the processor can issue the chained prefetches midway through the new jump interval, eliminating the stalls. We call this *staggered chain jumping.*[4] A longer jump interval, however, increases the chances of the jump-pointer prefetches being too early or damaging, and also increases the number of nodes at the beginning of the structure that are not prefetched at all.

Root jumping. This idiom is a variant of chained prefetching. It is applicable to short LDS where most of the LDS is accessed in the startup period and to dynamic LDS where updating jump pointers creates considerable overhead. In root jumping, an entire LDS has a single jump-pointer to the next LDS to be accessed. When an LDS traversal is begun, the jump pointer is used to prefetch the first node of the next LDS. Subsequently, the natural pointers of the second LDS are used to

[4]This method is not explicitly mentioned in [26], but was used in the prefetch codes we obtained from Roth.

issue prefetches to its nodes in lockstep with the traversal of the nodes of the first LDS. The use of the natural pointers is like chained prefetching and incurs serialization effects similar to chained prefetching.

Roth and Sohi described three different ways of implementing the above idioms – software, cooperative, and hardware. The software method uses a software queue structure to create jump pointers. The queue holds pointers to LDS nodes, and its size is the same as the jump interval. When an LDS node is traversed, a pointer to it is added to the queue. The head of the queue was accessed "jump-interval" iterations before this node, so the head of the queue is removed and its jump-pointer is set to point to the current node. The software method also inserts explicit software prefetch instructions for each jump-pointer and chained prefetch. In this paper we quantitatively evaluate the software method. The cooperative and hardware approaches are discussed in Section 4.2.4.

4.2.2 Qualitative Analysis of Processor-Side vs. Memory-Side Prefetching

We identify four factors that determine the performance differences between processor-side (represented by software jump-pointer prefetching) and memory-side schemes:

Instruction overhead

The processor-side schemes require the creation and possibly updating of jump-pointers. We can ignore the creation overhead since it is performed in the initialization phase of the application, but the overhead of updating the pointers for dynamic structures can be considerable. Staggered chain jumping also incurs such overhead because delaying the chained prefetches requires actions similar to updating jump-pointers. Finally, while the memory-side scheme requires a single prefetch command to be executed for each LDS, the processor-side schemes require a prefetch instruction to be executed for each LDS node that is not within the startup period. Thus, overall, the memory-side scheme has lower instruction overhead.

Unoverlapped latency in the startup period

As mentioned in Section 4.2.1, the processor-side schemes require a startup period where the initial LDS nodes are not prefetched. During this period, the LDS node accesses have their latency fully exposed. Therefore, for all processor-side schemes other than root jumping, the startup time is a function of the computation per node and the memory latency. For root jumping, we consider the startup period to be when the entire first LDS is worked on (since none of its nodes are prefetched); therefore, the startup time is also a function of the LDS length. The memory-side scheme's startup period is typically much smaller. If the prefetch command is issued just before the LDS traversal,

then accessing the very first node will require waiting for a round trip time to memory. Subsequent node accesses are part of the steady state which is analyzed later. If the prefetch command can be issued sufficiently early, then even the first node of the LDS will not incur stall time.

To better quantify the difference between the schemes, we present a simple "first-order" analytical model of the memory stall time incurred during the startup period. Let l be the average memory latency, L be the length of the LDS, c be the computation performed per node, and e be the time between when a prefetch command is issued and when the first LDS node is accessed (denoting how *e*arly the command is issued). Based on the above discussion, we get the following expressions.

For queue, full, and chain jumping, $Startup\ stall\ time \approx jump\ interval \times l$, where

$jump\ interval = \lceil \frac{l}{c} \rceil$ for queue, full, and regular chain jumping

$= \lceil \frac{2l}{c} \rceil$ for staggered chain jumping

For root jumping, $Startup\ stall\ time \approx L \times l$

For memory-side prefetching, $Startup\ stall\ time \approx max\{0, l - e\}$

The above equations show that for processor-side prefetching, the startup cost can be considerable (it is a function of the square of the memory latency for all but root jumping). In contrast, for the memory-side scheme it is at most equal to the memory latency and can be zero if the prefetch command can be issued early enough.

Steady state prefetch behavior

After their startup periods, both prefetching techniques enter a kind of steady state, where the processor may spend a certain amount of time waiting for a node to load and then works on the node. For the processor-side prefetching scheme, with queue, full, and staggered chain jumping, the jump interval can be adjusted to match the amount of work done per LDS node such that there is no memory stall time per node. Root jumping does not have this advantage because it prefetches nodes without using a jump interval. Instead, the next LDS to be accessed is always prefetched in lockstep with the current one, allowing overlap with the work for only a single node. An analogous observation applies to regular chain jumping as well. Thus,

For queue, full, and staggered chain jumping: $Average\ steady\ state\ stall\ time\ per\ node \approx 0$

For root and regular chain jumping: $Average\ steady\ state\ stall\ time\ per\ node \approx max\{0, l - c\}$

In the memory-side scheme, the prefetch engine traverses the LDS one node at a time, with a delay of one DRAM access time between two prefetches. Assuming the DRAM access time (say d)

is the longest stage in the entire path between the processor and memory and there are no prefetch command migrations, prefetched data will appear at the processor every d cycles. If a prefetch command needs to migrate, however, then the average time a prefetch command spends migrating per node, say m, needs to be added to this delay. If the computation time per node, c, is more than $d+m$, there is no memory stall time in the steady state. Otherwise, the steady state behavior depends on how early the prefetch command was issued. If the prefetch command was issued early enough, then the post-startup period may consist of two phases.[5] First, some nodes at the beginning of the LDS may have been prefetched before their demand accesses. But since the processor performs less work per node relative to the time between prefetch arrivals, it will eventually catch up to the prefetch engine and begin the second phase where each node sees a delay. This delay is the difference between the time between prefetch arrivals ($d+m$) and the computation time per node (c). Thus,

for memory-side prefetching,
if $c \geq d+m$, *Average steady state stall time per node* $\approx$ 0
if $c < d+m$,

$$\begin{aligned} \textit{Average steady state stall time per node} &\approx 0 \text{ for a few nodes at the beginning of the LDS} \\ &\approx d + m - c \text{ for the remaining nodes} \end{aligned}$$

In summary, for the steady state, full, queue, and staggered chain jumping do not see stall time. Root and regular chain jumping see stall time unless computation time per node is more than round-trip memory latency. Memory-side prefetching sees no stall time if the computation time per node is more than the sum of the DRAM access time and average per-node prefetch command migration time. Otherwise, it could see some stall time unless the prefetch command is issued early enough. Prefetch commands are more likely to be issued early enough to eliminate the stall time if the LDS is small.

Network, memory, and directory bandwidth

The processor-side scheme issues memory requests for every prefetch that misses in the cache. Most of these requests are sent instead of requests that the processor would normally send when accessing the LDS nodes. Some of these prefetches, however, are useless; therefore, the overall network, memory, and directory bandwidth requirement increases slightly for the processor-side scheme. The bandwidth requirement for the memory-side scheme could be higher or lower than for the base

[5] For simplicity, we continue to refer to this two phase period also as the "steady state."

case, depending on the number of LDS nodes traversed per prefetch command, missed coalescing opportunities, prefetch command migrations, and useless prefetches, as discussed in Section 4.1.2.

Summary

Table 5 summarizes the above analysis. Memory-side prefetching is expected to be better when the LDS being accessed are small (relative to the jump interval required), while processor-side prefetching is better when the LDS accessed are large and have little work done per node (relative to the DRAM latency and command migration overhead). For large LDS with a reasonably large amount of work done per node, the startup and steady state effects are negligible for both schemes. However, the instruction overhead of the processor-side scheme makes memory-side prefetching more attractive, especially if the LDS is dynamic or requires many prefetches per node (e.g., backbone and ribs structure). Also, if memory bandwidth is a limitation then memory-side prefetching could further improve application performance by reducing the system's bandwidth usage as long as the prefetch command migration rate is not too large. Processor-side prefetching does not reduce bandwidth usage, and generally increases it.

4.2.3 Results

We evaluate software jump-pointer prefetching for all of the benchmarks studied. To ensure a fair comparison, we obtained the code used in [26] to determine where to insert prefetches. (The prefetches used in the memory-side scheme are analogous.) For benchmarks where multiple jump-pointer prefetching idioms were applicable, we selected the idiom that provided the best performance on our architecture for each benchmark as summarized in Table 6. We also adjusted the jump intervals to achieve maximum performance. We did not add a prefetch buffer to the system as in [26] because the benchmarks exhibit little cache pollution. Also, the buffer would need to be visible to the coherence mechanism since this is a multiprocessor system. Note that the system in [26] is a uniprocessor system where this extra complexity is not required.

Comparing the MPF and PPF bars in Figure 3, we see that memory-side prefetching performs significantly better than processor-side prefetching for *em3d*, *health*, and *mst*, with a reduction in execution time of 20%, 20%, and 50% respectively. Processor-side prefetching performs significantly better than memory-side prefetching for *treeadd* with a reduction in execution time of 18%. Both schemes perform similarly for *bisort*, *perimeter*, and *tsp* (within 4%). The following uses the analysis from the previous section to explain these results.

As before, we do not discuss *bisort* or *perimeter* further since neither technique works well for

Scheme	Advantages	Where is it the best scheme?
Memory-side prefetching	(1)Lower instruction overhead (2)Shorter startup time (3)Possibly reduced bandwidth	(1) Small LDS (2) Large LDS with lot of work per node, where LDS is dynamic or requires many prefetches per node
Processor-side prefetching (Software jump-pointer prefetching	(1) Shorter steady state stall time, except for root or regular chain jumping	Large LDS with little work per node (unless system is bandwidth bound)

Table 5 Characterization and comparison of memory-side prefetching and processor-side prefetching

App.	Idiom	App.	Idiom	App.	Idiom	App.	Idiom
bisort	queue	health	full	perimeter	queue	tsp	queue
em3d	staggered chain	mst	root	treeadd	queue		

Table 6 Jump-pointer prefetching idiom implemented for each benchmark

the non-deterministic traversals in these benchmarks.

For *em3d*, the LDS are of moderate size, with a large amount of work done per node, so neither technique has significant steady state stall time. The memory-side scheme performs better here because processor-side prefetching incurs large instruction overhead from the staggered chain jumping technique, and also because the startup period has a negative effect.

The LDS used in *health* are dynamic, relatively small, and the amount of work done per node is also small. The primary reasons for the performance difference between the two schemes are the very significant startup period for the processor-side scheme, and also its overhead of maintaining the jump-pointers for the dynamic LDS. The memory-side scheme does suffer from some steady state stall time due to the small amount of work done per node, but prefetch commands for the LDS can be issued early, hiding most of it. Therefore, the memory-side scheme eliminates more of the memory stall time and outperforms the processor-side scheme for *health*.

Mst accesses many hash table buckets, so the LDS are very small and very little work is done per node. The structures are static and root jumping is used for the processor-side scheme, so instruction overhead and startup effects are negligible. However, root jumping suffers from steady state stall time, while the prefetch commands in the memory-side scheme can instead be sent out quite early, largely eliminating this. This allows the memory-side scheme to perform much better than the processor-side scheme for this benchmark.

For *treeadd*, the LDS accessed are very large, static binary trees which have little work performed

per node. The startup period and instruction overhead effects are quite small for the processor-side scheme, and it uses queue jumping which ideally has no steady state stall time. The memory-side scheme has significant steady state stall time because of the little work done on each node and the large size of the LDS. Therefore, the processor-side scheme outperforms the memory-side scheme for *treeadd*.

Finally, *tsp*'s LDS are large, static, and have a moderate amount of work done per node. Both prefetching techniques therefore perform similarly and reasonably well.

In summary, we find that memory-side prefetching is effective, but a combination of processor-side and memory-side prefetching is best. The third column of Table 5 can be used to decide which scheme to use for a specific LDS traversal.

4.2.4 Alternative Jump-Pointer Prefetching Implementations

Cooperative and hardware jump-pointer prefetching schemes augment the processor with hardware to overcome some of the problems with software jump-pointer prefetching. The cooperative scheme uses a hardware engine as described in [25] to perform the root jumping accesses and to perform chained prefetches (including generating the addresses of these prefetches). The engine views an LDS node read as producing a result that is consumed by the chained prefetch. It captures the producer-consumer relationship, and watches for values loaded by the producer. Upon seeing such a value, it speculates that the consumer will be executed and so issues a corresponding prefetch. This engine requires two tables and a queue to hold prefetch requests. One table holds information used to identify chained prefetches and the other holds the information necessary to speculatively generate the address for the chained prefetch. The pure hardware scheme augments the cooperative scheme with hardware that creates and updates jump-pointers and performs jump-pointer prefetches. The new hardware added for this scheme is a queue used for jump-pointer creation (analogous to the software queue described in Section 4.2.1) and a special register to hold the address of a node to prefetch. It also requires new flavors of load instructions to be used for the recurrent LDS node accesses. These special loads inform the hardware about the location of any extra space in each LDS node for the automatic placement of the jump-pointers.

We next qualitatively discuss the impact of the cooperative and hardware schemes. The cooperative approach only enhances the performance of root jumping and regular chain jumping; i.e., only *mst* in our benchmark suite. It has the same startup time as in the software case, but improves the steady state stall time. Since hardware performs the chained prefetches, the processor is not forced

to stall for the address generation of these prefetches (which are dependent on previous accesses). These prefetches themselves, however, are still sent in serial fashion, but are in parallel with the rest of the processor activity.

The following develops an analytical expression for steady state stall time for *mst* with the cooperative implementation (the only benchmark where the cooperative approach is applicable). Once the jump-pointer pointing to a list is prefetched, the hardware will traverse the list without interruption. It can overlap this traversal with computation and memory stall time for the entire predecessor LDS (since this is root jumping), plus the computation time for the LDS being prefetched. The last nodes in the LDS will incur more stall time than the first nodes, but we can still derive an expression for the average steady state stall time incurred per node. The *total* steady state stall time will be the time to traverse the list minus the time that is overlapped, as mentioned previously. The traversal time is the length of the LDS times the memory latency ($L \times l$). The computation and memory stall time for the previous LDS is the length of an LDS times the average computation per node plus the average memory stall time per node ($L \times (c + \mathit{stall\ time\ per\ node})$). The computation time for the current LDS is its length times the computation per node ($L \times c$). Finally, we combine these expressions, divide by the LDS length (to obtain the per node stall time), and simplify. We find that the average steady state memory stall time per node is $\approx \frac{l}{2} - c$. Thus, the average memory stall time per node is still a function of the round trip memory latency. We therefore expect that memory-side prefetching will continue to outperform even a cooperative approach for *mst* (although the gap may be reduced).

The hardware jump-pointer prefetching scheme does not have any further effect on the startup times or steady state memory stall times. Instead, it eliminates instruction overhead, which could reduce the performance gap between memory-side and processor-side prefetching for *em3d* and *health*. However, it will not eliminate the gap due to the startup stall time effect.

Overall, memory-side prefetching is expected to continue to outperform even the cooperative and hardware versions for some applications; however, the decision to implement any of these schemes must also consider relative hardware complexity.

4.2.5 Comparison with Prefetch Arrays

Recently, Karlsson et al. proposed a technique to reduce the startup stall time for jump-pointer prefetching [13]. This technique creates an array of pointers, called the prefetch array, pointing to the first few nodes in an LDS that do not have jump-pointers. Just before accessing the LDS,

prefetches are issued to all nodes pointed by the prefetch array, potentially hiding some latency for those nodes as well. Prefetch arrays are also used to allow more effective prefetching of trees with input-dependent traversals. Each node contains a prefetch array which holds pointers to all nodes at a distance equal to the jump interval, instead of a jump-pointer. When a node is traversed, prefetches for all elements indicated by its prefetch array are issued. The actual traversal path will be prefetched (since all possible traversal paths within the jump interval are prefetched), but a lot of nodes are prefetched unnecessarily. Finally, to reduce prefetch instruction overhead, a hardware block prefetch instruction is proposed that can trigger multiple prefetches in hardware.

We implemented prefetch arrays for *em3d*, *health*, *mst*, and *tsp*, the list-based benchmarks. The only benchmark sped up significantly by prefetch arrays is *health*, where memory stall time from the startup period is reduced. The execution time reduction over Base for *health* with prefetch arrays is 20% instead of the 11% when prefetch arrays are not used. (Memory-side prefetching shows a reduction of 29%.) For *em3d*, the large number of prefetches issued from the prefetch array interfere with the work performed on the first nodes in the list, *degrading* execution time relative to jump-pointer prefetching by 3%. For *tsp*, the LDS are large, and the additional overhead of maintaining the prefetch arrays is larger than the benefit of prefetching the extra nodes, so again execution time degrades by 3% (relative to jump-pointer prefetching). Finally, for *mst*, accessing a prefetch array often causes an additional cache miss. Also, many of the prefetch array accesses miss in the TLB, partially serializing the prefetches. These combine to limit the execution time reduction over the root jumping implementation to 4%.[6]

We did not implement prefetch arrays for the tree-based benchmarks, *bisort*, *perimeter*, and *treeadd* for the following reasons. For *treeadd*, the startup time is insignificant due to the large LDS and the traversal is deterministic; therefore, no significant additional benefit would be derived from prefetch arrays. The other two benchmarks have input-dependent traversals. However, the trees used in *bisort* are highly dynamic; updating the prefetch arrays would be impractical and incur a very large overhead. The trees in *perimeter* are quad-trees, and Karlsson et al. found that too many unnecessary prefetches are issued for *perimeter* to receive any significant performance benefits.

We also did not implement the hardware block prefetch instruction, but do not expect that to make a difference to the above results.

In summary, we find that prefetch arrays did not give consistent benefits over the jump-pointer

[6]Karlsson et al. indicate a significant performance benefit over jump-pointer prefetching for *mst*. We believe this difference in our results arises because they performed their comparison with queue jumping rather than root jumping.

prefetching schemes of Roth and Sohi (only one benchmark showed a somewhat significant gain of 10% but half of the benchmarks showed or are expected to show a degradation). Nevertheless, the technique does have potential for hiding jump-pointer prefetching startup time for some cases. A more detailed characterization of those cases and a combined approach with memory-side prefetching is a promising direction for future work, but outside the scope of this thesis.

5 Related Work

There has been much speculation about the future of merged DRAM-logic. Burger et al. developed the DataScalar architecture [3], a multiprocessor PIM system for running uniprocessor applications. The PIMs in the system all run the full application on the entire input data set, but they act as very intelligent prefetch engines for one another by sending local data to the other processors as it is needed. While this generates excellent prefetching characteristics, it appears to be an inefficient use of the hardware because of the large amount of redundant computation.

The Berkeley IRAM group has proposed integrating a vector processor with DRAM. Their Vector IRAM architecture [14] is targeted towards media processing applications and other applications that operate on streams of data. Their work is not focusing on LDS or on systems with multiple IRAM chips.

The Active Pages [21] and FlexRAM [12] projects seek to effectively integrate a number of simple processing elements onto a DRAM chip. The processor can instruct the processing elements to perform a set of parallel functions on data on the chip. These projects are targeted to applications with a large amount of data parallelism, whereas LDS traversal discussed in this paper is inherently a serial operation.

Impulse is an intelligent memory controller capable of remapping physical addresses to improve performance of applications with irregular data access patterns [6]. Impulse is capable of prefetching data, but only implements next-line prefetching. Also, it does not send the data to the processor, but rather buffers it at the memory controller.

In the area of prefetching, there has been a large amount of work done, both for hardware and software prefetching techniques. The traditional work in this area has focused on prefetching of regular data structures such as arrays [7, 8, 24, 20].

More recently, a number of prefetching techniques for LDS have been proposed. We have already discussed the work in [26, 13] in Section 4. SPAID [16] was an attempt to tackle pointer prefetching by issuing prefetches for pointers that are passed as arguments to functions. The prefetch is overlapped with the function call to hide some latency. However, this provides little coverage for LDS based applications since functions called on an LDS will have only a single node prefetched.

Zhang et al. discuss a multiprocessor system where a coprocessor sends LDS prefetch requests on behalf of the main processor [29], but they do not show much gain (average of 8%) from the scheme.

Luk et al. proposed greedy prefetching, where the processor prefetches all successor pointers of the current LDS node [17]. However, the prefetches can be overlapped with the work for only a single

node. They also proposed LDS linearization, which involves allocating LDS nodes in a contiguous region of memory so that address prediction for LDS nodes is extremely easy and accurate. However, this only applies in a limited number of applications. Luk et al. extend this work in [18], where dynamic data movement is supported to make linearization applicable in more cases, but truly dynamic LDS still remain a problem. Other research has been performed on placement of data with regard to improving spatial locality (but not with respect to enhancing prefetching). Calder et al. provide a detailed algorithm for data placement in order to reduce all classes of cache misses for regular applications [4]. Chilimbi et al. focus on trees, and introduce an allocator and dynamic update function to pack multiple nodes from subtrees into a single cache line [9].

Luk et al. also proposed using jump-pointers [17], which were further developed by Roth et al. [26] and discussed in Section 4. Roth et al. first proposed dependence-based prediction [25], a more general-purpose version of a scheme developed by Mehrotra et al. [19]. The dependence-based prediction scheme is the same as the hardware part of cooperative jump pointer prefetching described in Section 4.2.4. This scheme can result in prefetches issued ahead of the processor's demand accesses, but the prefetches are all serialized. The later work by Roth and Sohi [26] supersedes dependence-based prefetching.

Another class of schemes use past access patterns to predict future accesses, including LDS accesses. These are known as correlation-based schemes. Alexander et al. use an SRAM buffer in DRAM chips to hold blocks of prefetched data [1]. The prefetches are based upon previous accesses to DRAM. For a given cache line, the address of the next request made to the DRAM is recorded in a table. When a line is accessed, the mechanism prefetches a limited number of previous successors. Joseph et al. propose a similar scheme, but place the prefetching hardware between the first and second level caches [11]. Finally, Bekerman et al. propose improvements to these schemes by introducing confidence mechanisms to improve the accuracy of predictions, and by introducing pollution reduction structures [2]. All of these schemes are limited in that large structures cannot be represented. The processor-side schemes also increase bandwidth requirements significantly since multiple prefetches may be issued for each line accessed. Finally, these schemes do not attempt to get far enough ahead of the processor to hide all of the memory latency.

Concurrently with our work, Yang et al. have developed a memory-side prefetching scheme similar to ours, but for a uniprocessor system [28]. They propose a prefetch engine at each level of the memory hierarchy that can only handle simple linked lists. They compare the results of their scheme to the dependence-based prediction scheme of Roth et al. [25], but not the more effective jump-pointer

schemes that we have examined. They find that memory-side prefetching always outperforms their processor-side scheme for their applications. Their analysis with a microbenchmark indicates that memory-side prefetching is better with less computation per node. We see different results since we compare with more aggressive processor-side schemes, and we give a qualitative framework for when the processor-side schemes or the memory-side schemes should be used for best performance.

6 Conclusions and Future Work

As merged DRAM-logic technology matures, PIM (processor-in-memory) chips are expected to become widespread. Applications whose memory segments do not fit within one such chip can take advantage of the technology by building systems out of multiple such chips. We have examined one way to exploit added intelligence to the DRAM in a multiprocessor PIM system. Our focus was speeding up applications which use linked data structures, an important but traditionally difficult set of applications to accelerate. We propose a prefetch engine close to memory. The engine attempts to perform a traversal of the LDS before its use at the remote or local processor, thereby prefetching the data for the processor. Applications with fixed traversal paths for their LDS, or traversal paths with limited dependence upon the contents of the LDS nodes, are sped up substantially by this technique (reduction in execution time of 19% to 62%).

We compared the proposed memory-side prefetching scheme to a state-of-the-art processor-side prefetching scheme. We found that the memory-side scheme outperforms the processor-side scheme for two classes of applications: (1) where the LDS are small compared to the round-trip memory latency (these incur high startup time for processor-side prefetching), and (2) where the LDS is dynamic or requires many prefetches per node (these incur high instruction overhead for processor-side prefetching). A technique for overcoming the high startup overhead with processor-side prefetching, prefetch arrays, is only able to significantly help in one of the applications studied and degrades performance slightly in some cases. The processor-side scheme outperforms the memory-side scheme for large LDS with little work per node. This is because the rate of prefetch transfers for the memory-side scheme is limited by DRAM latency and prefetch command migration frequency; if this rate is low relative to the work done per LDS node, then memory-side prefetching sees stall times in the steady state. Thus, we conclude that a combination of memory-side and processor-side schemes would prove most effective as a general technique; our characterization can aid in choosing the appropriate scheme for a specific LDS.

For future work, we will explore a more general prefetch engine that can traverse LDS that have non-deterministic paths that were not captured by the prefetch command implemented in this thesis.

References

[1] Thomas Alexander and Gershon Kedem. Distributed Prefetch-buffer/Cache Design for High Performance Memory Systems. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, 1996.

[2] Michael Bekerman et al. Correlated Load-Address Predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[3] Doug Burger, Stefanos Kaxiras, and James R. Goodman. DataScalar Architectures. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[4] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-Concious Data Placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[5] Martin C. Carlisle and Anne Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the 6th Principles and Practice of Parallel Programming*, 1995.

[6] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, and Lixin Zhang. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[7] Tien-Fu Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[8] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 1995.

[9] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-Concious Structure Layout. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, 1999.

[10] Bruce L. Jacob and Trevor N. Mudge. A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[11] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[12] Yi Kang et al. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the 1999 International Conference on Computer Design*, 1999.

[13] Magnus Karlsson, Fredrik Dahlgren, and Per Stenström. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000.

[14] Christoforos E. Kozyrakis and David Patterson. A New Direction for Computer Architecture Research. *IEEE Computer*, November 1998.

[15] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanović, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, September 1997.

[16] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software Prefetching in Pointer- and Call-Intesive Environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[17] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[18] Chi-Keung Luk and Todd C. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guranteeing the Safety of Data Relocation. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

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

[20] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[21] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[22] V. S. Pai, P. Ranganathan, H. Abdel-Shafi, and S. Adve. The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors. *IEEE Transactions on Computers, special issue on caches*, February 1999.

[23] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. *RSIM Reference Manual.* Rice University.

[24] Shlomit S. Pinter and Adi Yoaz. Tango: a Hardware-based Data Prefetching Technique for Superscalar Processors. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[25] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

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

[27] Steven Cameron Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.

[28] Chia-Lin Yang and Alvin R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. *To appear in Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[29] Zheng Zhang and Josep Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

Appendix A: A Prefetch Command for Memory-Side LDS Prefetching

In our implementation of memory-side prefetching, the prefetch command consists of the op code and the address of the first node of the LDS to be traversed (as in conventional software prefetching). Additionally, the command is augmented with a two word (64 bits)[7] description of the LDS and traversal path; options are described in Section 2.3. This appendix describes an encoding for these options. As discussed in Section 2.2, these options and encoding are not intended to be universally applicable, but are general enough for most of the studied benchmarks, which represent a wide range of LDS algorithms. There are two types of prefetch commands, one for deterministic traversal paths (Figure 5(a)), and one for paths with a limited dependence on the data in the LDS (Figure 5(b)). For both types of commands, the size of a node, in words, is included to allow support for prefetching large nodes.

The format for deterministic traversals includes pointer and rib fields, which are the offsets into a node for the successor pointers and rib pointers. For a list, a single successor pointer is used, and for trees multiple successor pointers are used. The command also holds the number of valid successor and rib pointers in each node (maximum of five successors and three ribs). Note that although the prefetch engine is targeted at LDS prefetching, it also supports a zero successor pointer count. This could be used to prefetch arrays or to perform other block data transfer prefetches.

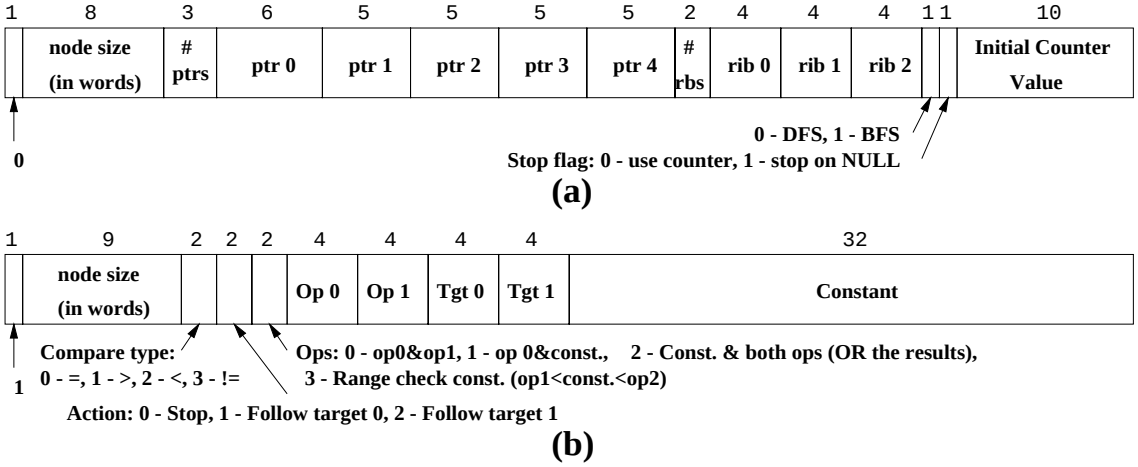


Figure 5 The format of a prefetch command. (a) Format for deterministic traversals. (b) Format for input-dependent traversals. The numbers above each field denote the number of bits in that field.

[7] We model a 32-bit architecture. A 64-bit architecture would provide the flexibility to encode even more options.

For this type of command, the order of traversal, depth-first (DFS) or breadth-first (BFS), is indicated by a single bit field. For lists, this field is irrelevant since there is only one possible path of traversal. Finally, the stop flag instructs the prefetch engine on when to end a traversal. When one, the traversal will not stop until the entire LDS has been prefetched. When the stop flag is zero, the traversal will stop when a given number of nodes have been traversed. This number is placed in the initial counter value field.

The other type of prefetch command is one that uses the result of one or two comparisons involving LDS data to determine the traversal path. The comparisons performed are governed by the compare command fields, compare type, action, and operands. The compare type field specifies the simple arithmetic comparison to perform, the action field specifies the action to perform on a successful comparison, and the operand field specifies the operands to compare (from two fields from the current LDS node and a 32-bit constant). The possible actions to perform after doing a comparison are to stop the traversal or to continue the traversal by following one of the possible successor pointers. The default action (i.e., the action to perform on an unsuccessful comparison) is to follow the first target pointer. The operand and target fields are offsets into the current LDS node and specify the comparison operands and possible successor pointers (targets), respectively. The constant field holds a full 32-bit constant for use in comparisons. Using 32-bits allows pointer comparisons which are useful when searching for specific nodes in an LDS.