The Impact of Timeliness for Hardware-based Prefetching from Main Memory

Wayne A. Wong and Jean-Loup Baer
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

email {waynew,baer}@cs.washington.edu

UW-CSE-02-06-03

Abstract

Among the techniques to hide or tolerate memory latency, data prefetching has been shown to be quite effective. However, this efficiency is often limited to prefetching into the first-level cache. With more aggressive architectural parameters in current and future processors, prefetching from main memory to the second-level (L2) cache becomes increasingly more important. In this paper, we examine the impact of hardware-based prefetchers at the L2-main memory interface on the performance of an aggressive out of order superscalar processor.

First, we show the importance of timeliness by simulating prefetch oracles with perfect coverage and accuracy. We show that in order to approach completely hiding the memory latency even under these perfect conditions, prefetches must be initiated more than one L2 cache miss ahead. For some applications, this required prefetching distance can be as much as four misses ahead.

Second, we simulated some previously proposed hardware prefetchers and investigated methods of extending their prefetching distances. Strided miss streams are predictable and easily prefetched with strided stream buffers. Applications interleave these miss streams to provide natural prefetching distances greater than one miss ahead. Miss streams with irregular address patterns are more difficult to prefetch for and have led to the proposal of other prefetching strategies, in particular address-correlation prefetching and linked-data structures (LDS) prefetchers. Neither of these is generally effective at the L2-main memory level. We have extended the LDS prefetcher into LDS stream buffers structures and propose a hybrid stream buffer prefetcher that works as well as strided stream buffers for scientific applications and yields a slight benefit for integer applications.

1 Introduction

For many classes of applications, the disparity between memory and processor speeds is the main impediment to sustained high-performance. Current technological trends and recent microarchitectural advances often have contradictory consequences in that respect. For example, the faster increase in CPU clock rates relative to the decrease in memory access times results in (relatively) longer service times from memory while extensions in the capacity of the cache hierarchy lower the amount of cache misses and, hence, the number of memory accesses. Similarly, the increasing exploitation of instruction level parallelism induces the misses to occur at smaller (real) time intervals while techniques to hide partially or tolerate memory latency such as lock-up free caches, dynamic scheduling, multithreading, and prefetching alleviate the consequences of long memory latencies. Nonetheless, it is apparent that reducing the memory latency for those applications that have a significant number of cache misses percolating to main memory is a prerequisite for achieving high performance.

In this paper, we focus on one of these techniques, namely hardware prefetching. While applicable at several levels in a system, we deal specifically with prefetching at the interface between the second-level cache (L2) and main memory. At this memory hierarchy boundary, caches misses have the most detrimental performance effects. We restrict ourselves to data prefetching because instruction cache misses are much less frequent at that level.

Efficient prefetching should be *accurate*, i.e., what is prefetched should be used, *timely*, i.e., the prefetched data should be in the cache or a prefetch buffer before it is referenced by the CPU, and done with high *coverage*, i.e., a large proportion of cache misses should be removed. Prefetching at L2 is more challenging than at the closer first-level (L1) because (1) predictions might not be as accurate since the L2 reference stream is filtered by the L1 cache and may carry less information, and (2) the miss latency is at least one order of magnitude longer and, therefore, prefetching must be initiated much earlier making timeliness more difficult to achieve. Finally, attempting to obtain high coverage at the expense of increased inaccuracy might result in over occupancy of the memory bus, the single channel where data flows between main memory and L2.

After reviewing previous work on hardware prefetching that, until now, has been mostly directed to the L1-L2 interface (Section 2) and describing the processor-memory hierarchy model and the methodology that we will use in our experiments (Section 3), we look at the potential margin of improvement that we can achieve assuming perfect accuracy and coverage. In Section 4 we use these oracles to show that the prefetch distance must often be significantly greater than one if memory latencies are going to be significantly hidden. In Section 5, we measure the performance of several hardware prefetchers. As could be expected, stream buffers are efficient for strided data structures, but linked data structures (LDS) prefetchers are, in general, of little or no help at the L2 level. We propose and evaluate LDS stream buffers as well as a hybrid predictor that combines regular and LDS stream buffers and provides the overall best performance. We summarize and conclude in Section 6. .

2 Related Work

Prefetching techniques can be categorized along two axes. The first is whether prefetching will be activated by a hardware mechanism, the topic of this paper, or by software [11], or a combination of both. The second axis is the type of data or address stream that is the target of prefetching. The goal of the simpler initial prefetching schemes was to take advantage of spatial locality. This was extended to data streams that exhibit regular strided addresses. However, stride prefetchers were found to be ineffective for applications with more random address patterns such as those that use linked data structures. Hence,

memory hierarchy		micro architecture	
L1 inst	perfect	inst queue size	32
L1 data	16 KB, 4-way, 32 B lines, 1 cycle latency	fetch size	8
L2	unified; 256 KB, 8-way, 64 B lines; write back, 10 cycle latency	branch predictor	perfect
cache ports	8	issue width	8
itlb	perfect	ruu size	128
dtlb	perfect	load-store queue size	128
memory latency	85 cycles	commit width	8
memory bus	16 B wide with 7.5 cycle/transfer	FUs: int alus	8
prefetch requests	16 outstanding	FUs: int mul	4
prefetch buffer latency	10 cycles	FUs: fp alu	8
		FUs: fp mult	4

Table 1. Simulator parameters.

prefetchers targeting applications with linked data structures became a topic of interest. Finally, correlation prefetchers make no assumptions on the type of data streams.

Increasing the cache line size is an implicit form of prefetching that has been extensively studied [14]. The earliest form of explicit prefetching is the one-block look-ahead (OBL or "next line") strategy [20] which, on a cache miss, not only loads the missing line but also the subsequent (address-wise) cache line. Sequential prefetching [6] extends OBL by allowing more than one line to be prefetched and by controlling the prefetching rate based on usage. The idea behind the original stream buffers [9] was to use the same "next line" strategy but the prefetched lines were stored in buffers distinct from the cache and dedicated to particular streams. These stream buffers were continuously filled as their data was consumed, thus helping to maintain the initial prefetching lead time. These unit stride stream buffers, and the related reference prediction tables [5], were extended to recognize non-unit strides [12, 7].

Recent prefetching strategies have targeted applications using LDS. These strategies can be separated into two classes. The first class prefetches each link sequentially by using either a cache assist that detects the execution of recurrent loads and prefetches accordingly by chasing pointers before they are accessed by the original program [17] or a separate execution engine [23, 24, 25], requiring user-compiler interaction, that executes slices of the original program in a look-ahead fashion. The second class augments the application's LDS to provide addresses (e.g., jump-pointers) further down the link chain and prefetches according to the jump pointer, thus potentially increasing the prefetch distance [18]

Correlation predictors, originally proposed in [15], build correlations consisting of (source, target) tuples between events such as consecutively referenced cache lines. Upon recognition of the source, its correlated target becomes a prefetch candidate. In the shadow-directory [3] and chain prefetching [21] schemes, the correlation target occupied a field in the metadata of the source cache line. More recent proposals use a separate correlation prediction table and have investigated different priority and target number policies [1, 8] as well as the combination of correlation and other prefetchers [8, 19].

While it is important to use the appropriate prefetcher for the targeted stream, it is equally important to achieve timeliness while retaining accuracy. Strategies to throttle the issuing of prefetches include control speculation [5, 13] and dynamically increasing or decreasing the amount being prefetched [6, 7].

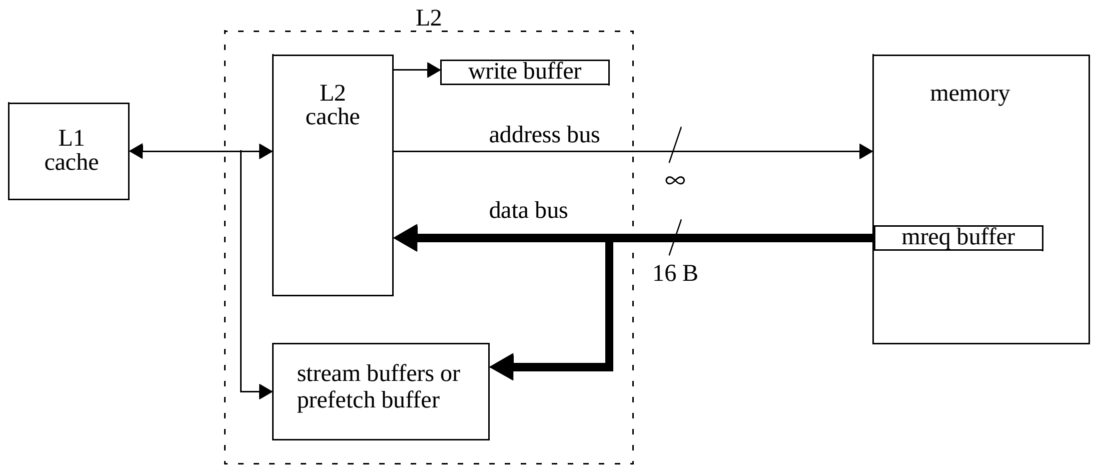


Figure 1. Memory hierarchy. This diagram shows the general memory hierarchy simulated. The L2 prefetch buffers are searched in parallel with the L2 cache. The L2 has infinite write buffers and memory address request bandwidth. The returning data bus width is 16 bytes. Returning prefetch data are buffered by the mreq buffer while they wait for the data bus.

3 Methodology

3.1 System Model

The prefetchers that we evaluate are hardware assists to a system that is relatively aggressive at the microarchitecture level. Its main memory bandwidth is idealized and provides constant latency at the device level. But the data communication between main memory and the cache hierarchy is modeled in a realistic fashion.

We used the Simplescalar framework [2] for our evaluations. We modelled an out-of-order core with the parameters listed in Table 1. The salient features are: 8-wide issue, an 128-entry Register Update Unit (RUU) [22], and a 128-entry load-store queue. The cache hierarchy has a perfect L1 instruction cache and perfect TLBs since we are mostly interested in the benefits of data prefetching at the L2 level. The L1 data cache is 16 KB, 4-way set associative with 32 byte lines, and an access latency of 1 cycle. Corresponding parameters for the L2 cache are 256 KB, 8-way, 64 byte lines, and 10 cycle access latency.The modelled main memory subsystem is shown in Figure 1. We made two optimistic assumptions: constant device latency and unbounded addressing bandwidth, yielding an address request plus device latency of 85 cycles. The return memory data bus adds an additional 7.5 cycles for each 16 bytes in the cache line. In the absence of bus contention, an L2 cache miss of 64 bytes has a total latency of 115 cycles. In the case of overlapping requests (contention), data waits for the data bus at the memory device level. Requests are queued in a 16-entry memory request buffer (*mreq* in Figure 1). Buffering might occur in the base case (no prefetching) because L2 is lock-up free and it will certainly happen when prefetching is activated and more than one cache line is prefetched. Demand misses have priority over prefetched data which, itself, is enqueued in a FIFO manner. These simplifying assumptions still allow us to evaluate and compare various prefetching techniques without dwelling too much in the details of specific memory devices and memory controller scheduling.

Data is prefetched in a separate buffer (or stream buffers) thus avoiding cache pollution. When an L1 miss occurs, L2 and the prefetch buffer are searched concurrently. If the data is not in L2 but is in the

class	application	source	startup insns (M)	insns (M)	description
integer	em3d	Olden	30	138	electromagnetic problem; 6000 nodes, arity 10
	health	Olden	3	200	health care simulation; 7 levels
	mcf-light	shareware	121	200	network simplex code version 1.1
	mst	Olden	1337	116	minimum spanning tree algorithm; 2048 nodes
	parser	SPEC 2000	600	200	dictionary lookup; ref input
	vpr	SPEC 2000	400	200	placement part of benchmark; ref input
floating-pt	hydro2d	SPEC 95	400	200	navier stokes equations; training input
	su2cor	SPEC 95	1000	200	monte-carlo method; training input
	swim	SPEC 95	600	200	shallow water equations; training input

Table 2. Benchmarks. For each application, the name, suite source, number of initial instructions ignored (startup insns), simulated instructions after startup phase (insns), and application description are listed.

	IPC baseline	IPC ideal L2	speedup	L2 MPI
em3d	0.631	3.508	5.559	0.0282
health	0.429	2.231	5.200	0.0188
mcf-light	0.181	1.080	5.967	0.0599
mst	0.319	2.613	8.191	0.0694
parser	1.531	3.521	2.300	0.0036
vpr	1.249	4.030	3.227	0.0083
hydro2d	1.172	5.796	4.945	0.0150
su2cor	2.414	5.204	2.156	0.0062
swim	1.279	3.606	2.819	0.0078

Table 3. Ideal L2 cache speedups. This table lists the IPC for both the baseline and ideal L2 models, number of instructions simulated, number of L2 misses, L2 MPI, and ratio of the ideal $\text{IPC}_{\text{ideal L2}}$ to the baseline's.

buffer, it is moved from the buffer to L1 and to L2. The latency is the same as for an L2 hit. Under these conditions, the effectiveness of prefetching will be mostly driven by the impact of accuracy, coverage, available memory bandwidth, and the timeliness of prefetches.

3.2 Benchmarks

The benchmark suite that we selected (Table 2) is drawn from the Olden [16], SPEC 2000, and SPEC 95 benchmark suites, to which we added a public network simplex solver. They were chosen because they have a significant number of L2 misses and because they represent a mix of applications exhibiting regular and irregular data access patterns.

To motivate the importance of L2 misses, we show (Table 3) the IPC obtained when running these applications on the base architecture from Table 1 ($\text{IPC}_{\text{baseline}}$) and a system with an "ideal" L2 ($\text{IPC}_{\text{ideal L2}}$) where all L1 misses hit in L2. The speedups of the ideal model range from 2 to over 8 depending on the application, thus showing that there is an opportunity for impressive performance improvements if one could remove L2 miss latencies (the next section will shed more light on this problem).

Because some of the prefetching strategies studied later in this paper use per-instruction classification, we estimated the working set of static instructions that generate L2 misses. To do so, we

application	IPC ideal L2	IPC ideal cpu	min (IPC ideal L2, IPC ideal cpu)	ideal speedup	ideal prefetch distance
em3d	3.508	1.182	1.182	1.8740	3.83
health	2.231	1.769	1.769	4.1234	3.83
mcf-light	1.080	0.557	0.557	3.0754	3.83
mst	2.613	0.480	0.480	1.5046	3.83
parser	3.521	9.163	3.521	2.2998	1.47
vpr	4.030	4.005	4.005	3.2068	3.83
hydro2d	5.796	2.217	2.217	1.8920	3.83
su2cor	5.204	5.363	5.204	2.1558	3.72
swim	3.606	4.280	3.606	2.8194	3.23

Table 4. Estimated prefetch distances. This table lists the estimated average prefetch distance required to tolerate a L2 miss penalty (115 cycles).

recorded the addresses (PCs) of the instructions generating L2 misses and kept them in a fully-associative buffer using LRU replacement. All applications generated a very high percentage of their misses (over 99%) from the 16 most recent PC's generating misses, except for *vpr* (the 99% figure is obtained with 64 PCs). Thus, a prefetching strategy that records per PC information for L2 misses in some form of hardware "table" should not require a large number of entries.

4 Prefetch Distance

4.1 Estimating the prefetch distance

In the previous section, we gave an indication of the IPC that could be achieved assuming a perfect L2. Another way of stating the same results is to consider that we performed prefetching with total coverage, complete accuracy, and perfect timeliness. Under the assumptions of our machine model, we can certainly simulate an oracle with 100% coverage and accuracy but complete timeliness may not be achievable. Even if we assume that all memory requests and memory accesses at the DRAM level can be totally overlapped with the transmission of data to L2, we need to serialize data transmissions on the bus. Thus, a minimum execution time is

$$\text{minimum execution time} = \text{number of L2 misses} \times \text{bus transfer time}$$

where *bus transfer time* is the time to transfer one cache line from the DRAM to L2. A second measure of the ideal IPC is therefore

$$\text{IPC ideal cpu} = 1 / (\text{L2 MPI} \times \text{bus transfer time})$$

where L2 MPI is the number of L2 misses per instruction. The "ideal" IPC then will be the minimum of the $\text{IPC}_{\text{ideal L2}}$ from the previous section and the above $\text{IPC}_{\text{ideal cpu}}$ (Table 4 shows these IPCs as well as the maximum speedup achievable). In some cases, the $\text{IPC}_{\text{ideal}}$ is more conservative than earlier calculations as the serialization of the memory bus becomes the bottleneck in those idealized cases.

In the above ideal cases, prefetching was initiated as early as possible and there was no consideration of the events that could trigger the requests. In practice though, we need to determine when to prefetch. For the purposes of this paper, we define the *prefetch distance* in terms of what would have been L2 misses, i.e., remaining L2 misses or hits in the prefetch buffer. For example, a prefetch distance of 1 means that on an L2 miss or an access to a prefetched line in the prefetch buffer, we initiate a prefetch to the line that would be the next one to miss. A prefetch distance of 2 would initiate the prefetch for the miss

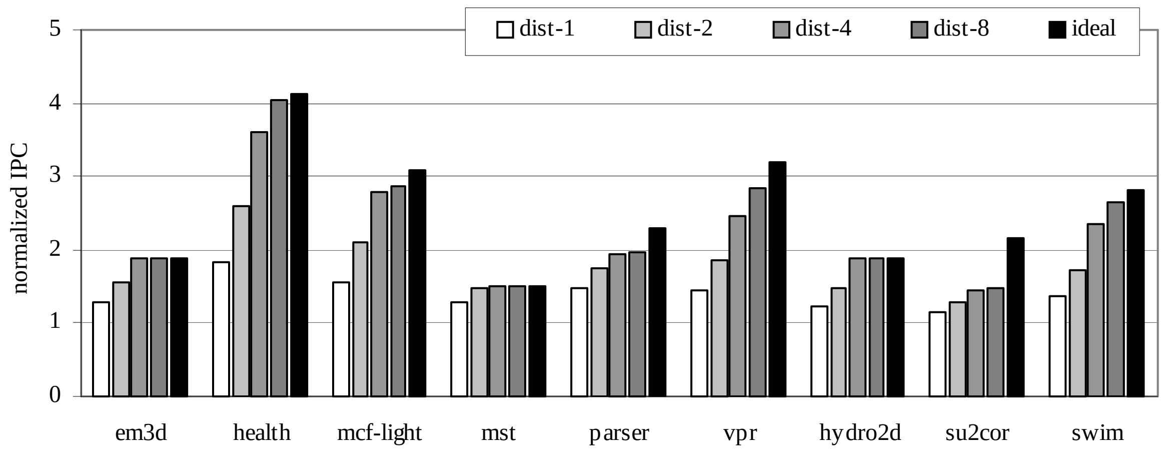


*Figure 2. Oracle prefetching.*This graph shows the performance of the oracle prefetcher with various prefetching distances.

after the next miss (or successful prefetch).

If we initiated prefetches at the occurrence of what are L2 misses in the baseline case, we can give an estimate for the average prefetch distance required to completely hide the memory latency as

$$\text{estimated required prefetch distance} = \text{L2 miss latency} \times \text{L2 MPI} \times \text{IPC ideal}$$

This estimate is neither a lower nor an upper bound since on one hand, $\text{IPC}_{\text{ideal L2}}$ can never be reached, and on the other hand $\text{IPC}_{\text{ideal cpu}}$ does not take into account possible overlaps between consecutive memory requests.

In the last column of Table 4 we show the *estimated required prefetch distance* for the various applications. These estimates indicate that prefetch distances greater than one, and sometimes as large as 4, would be needed to completely hide a main memory latency of 115 cycles. These figures indicate the paramount importance of timeliness in prefetching into L2.

4.2 Oracle prefetching

To further investigate the impact of the prefetch distance and to assess potential margins of improvement, we supplemented the baseline system (no prefetching) with oracles having perfect coverage and accuracy for several prefetching distances. The results are shown in Figure 2, where the IPC's of the oracles with varying prefetch distances are plotted in a normalized fashion against the baseline case. For all applications there is a medium (14% in *su2cor*) to significant (82% in *health*) improvement over the baseline if one could prefetch perfectly one L2 miss ahead (dist-1). Note that these speedups are only 44% to 86% of the ideal even with our optimistic assumptions. What is more interesting, though, is that this improvement is almost doubled in all cases when the prefetch distance is 2 (dist-2). In more than half of the cases though, there is still a significant gap between the ideal IPC and the oracle with a prefetch distance of 4 (dist-4) which is greater than our estimated required prefetch distance. One reason for this mismatch is that our estimated prefetch distance is based on averages while the latency tolerance of our system is time-varying. Prefetching earlier than required does not provide additional benefit over prefetching just in time. Similarly, the prefetching rate may not be uniform and can lead to bursty prefetching which generates contention for the bus and increases the latency for the prefetched data.

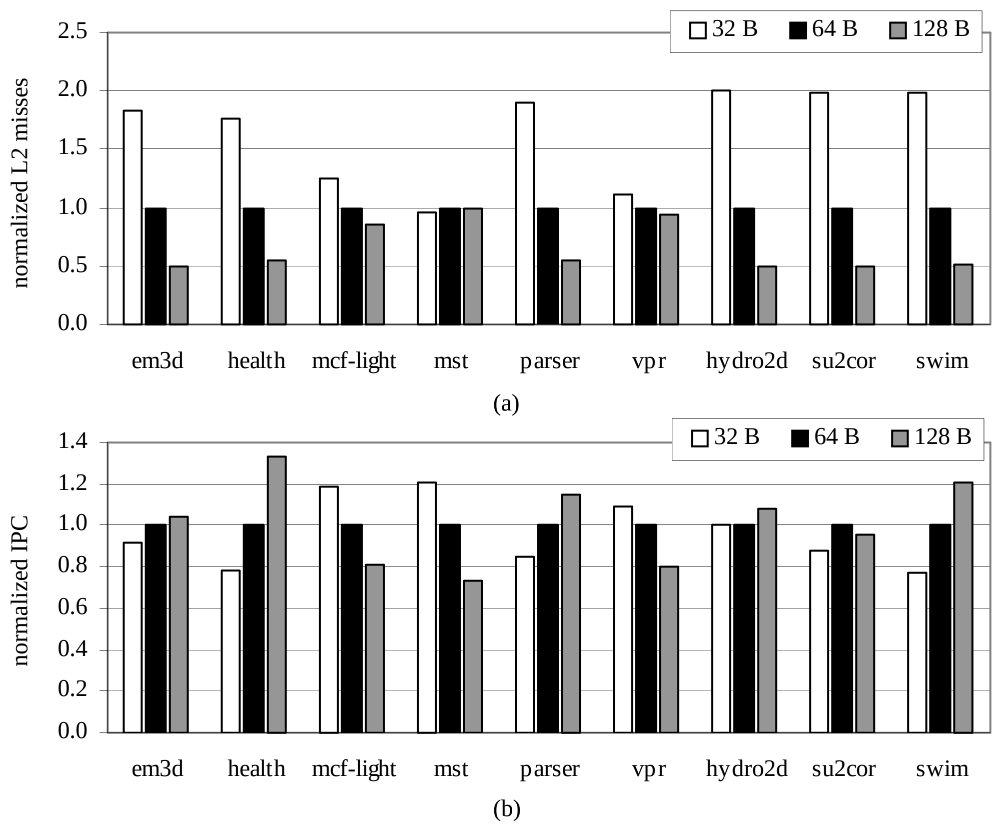


Figure 3. Performance with various L2 line sizes. Figure (a) shows the relative change in L2 cache misses which have been normalized to the number of misses with a 64 byte line size. Figure (b) shows the resulting overall normalized IPC for various L2 line sizes. The IPC values have been normalized to the IPC with 64 B L2 line size.

From these experiments, we can confirm that prefetching strategies will need to be able to prefetch at L2 more than one miss ahead in order to significantly hide the memory latency. Of course, this is in addition to being accurate and to cover most misses. As the memory latency increases, the prefetching distances will also need to be further increased. Overall, this is a challenge that cannot be met adequately for all applications by the prefetching techniques that we evaluate in the next section.

5 Hardware Prefetching Strategies

5.1 Implicit Prefetching

Because L2 cache line sizes are larger than the size of the requested datum that caused the miss, a cache miss performs implicit prefetching. The impact of cache line fetch sizes has been extensively studied [14]. Thus, we limited ourselves to a set of experiments with the goal of finding the best line size for the baseline machine running the applications listed in Table 2.

The system we are simulating has a highly associative (8-way) L2 and therefore a larger line size will in general reduce significantly the miss rate (Figure 3a). However, because of the longer latency in transferring a larger line from main memory, increasing the cache line size does not necessarily translate into a higher IPC (Figure 3b). In the remainder of this study, we will consider a cache line size of 64 bytes, the best compromise for the baseline case.

5.2 Regular address streams

5.2.1 Per-instruction stream buffers

As mentioned in Section 2, stream buffers extend the concept of one-block-lookahead (OBL) prefetching. Upon recognition of misses that form a strided data stream, FIFO buffers are filled with additional lines from the same stream. The prefetch distance can be increased by prefetching more than one line at allocation time and can be maintained by keeping the FIFO buffer continuously full. Also, the interleaving of prefetching for several streams implicitly increases the prefetch distance.

In our simulations we implemented a per-instruction stride predictor [7]. L2 miss addresses are recorded in a PC-indexed table. A simple state machine detects whether the sequence of miss addresses for a particular PC forms a strided stream. Upon the detection, a stream buffer is allocated from a pool of buffers using an LRU replacement policy and prefetching for the newly formed stream buffer is initiated. On an L1 miss, L2 and the stream buffers are searched concurrently. On a hit to a stream buffer, the line is transferred to the L1 and L2 caches and a prefetch is initiated to fill the stream buffer whose line was consumed.

The prefetch distance can be increased by providing deeper stream buffers. A disadvantage of the greater depth is that initial additional bandwidth is needed when a stream buffer is allocated, thus potentially slowing demand fetches and prefetches for other streams. Therefore, rather than completely filling the buffer on allocation, we use incremental prefetching [7]. On allocation, only one buffer entry is prefetched. Whenever there is a stream buffer hit, the fetch size is doubled until the buffer's depth size is reached.

5.2.2 Experiments

Recall from Section 3.2 that, in general, 16 PC's at any instant generate the great majority of L2 misses. This observation allows us to have a small stride detection table of PC's that we fix at 32 entries. For the same reason, we limit our investigation to a maximum of 16 stream buffers but we will also examine the performance for 4 and 8 buffers. We will vary the depth of these buffers from 1 to 8.

Figure 4 summarizes the results of the simulations. The speedups over the baseline of systems with 4, 8, and 16 stream buffers with varying depths are shown for each application. For comparison purposes, the last bar for each application shows the speedup of an oracle of prefetch distance 1.

As was reported in [12], stream buffers are very effective for scientific applications (the rightmost three sets of bars in Figure 4). For these applications performance improves with the number of stream buffers until there are enough buffers to cover the PCs generating misses. Increasing buffer depth seems to be beneficial as long as sufficient buffers exist. For example, *su2cor* and *swim* do not benefit from increased buffer depths if there are only 4 stream buffers or less. Thus it is more important to have a larger number of shallow buffers to cover the distinct PCs generating misses rather than a smaller number of deep ones. To understand better why increasing depth is not as crucial, we show in Figure 5 the percentage of L2 misses whose latencies are completely hidden by prefetching. While the percentage grows with increasing depth, it does not grow at the rate that the buffer depth is increased. The increased number of hidden prefetches is counter balanced by the latencies of prefetches and demand fetches that are in-flight. The deeper buffers allow some prefetches to be initiated earlier and arrive in a more timely manner. However, because of inaccurate prefetching, the additional prefetches consume bandwidth that may delay demand fetches or other prefetches. Finally, we note that as soon as the number of buffers is sufficient, the performance is better than with an oracle of prefetch distance 1. This gain can be explained by the interleaving factor mentioned earlier; namely that even with a depth of 1, stream buffers prefetch at a

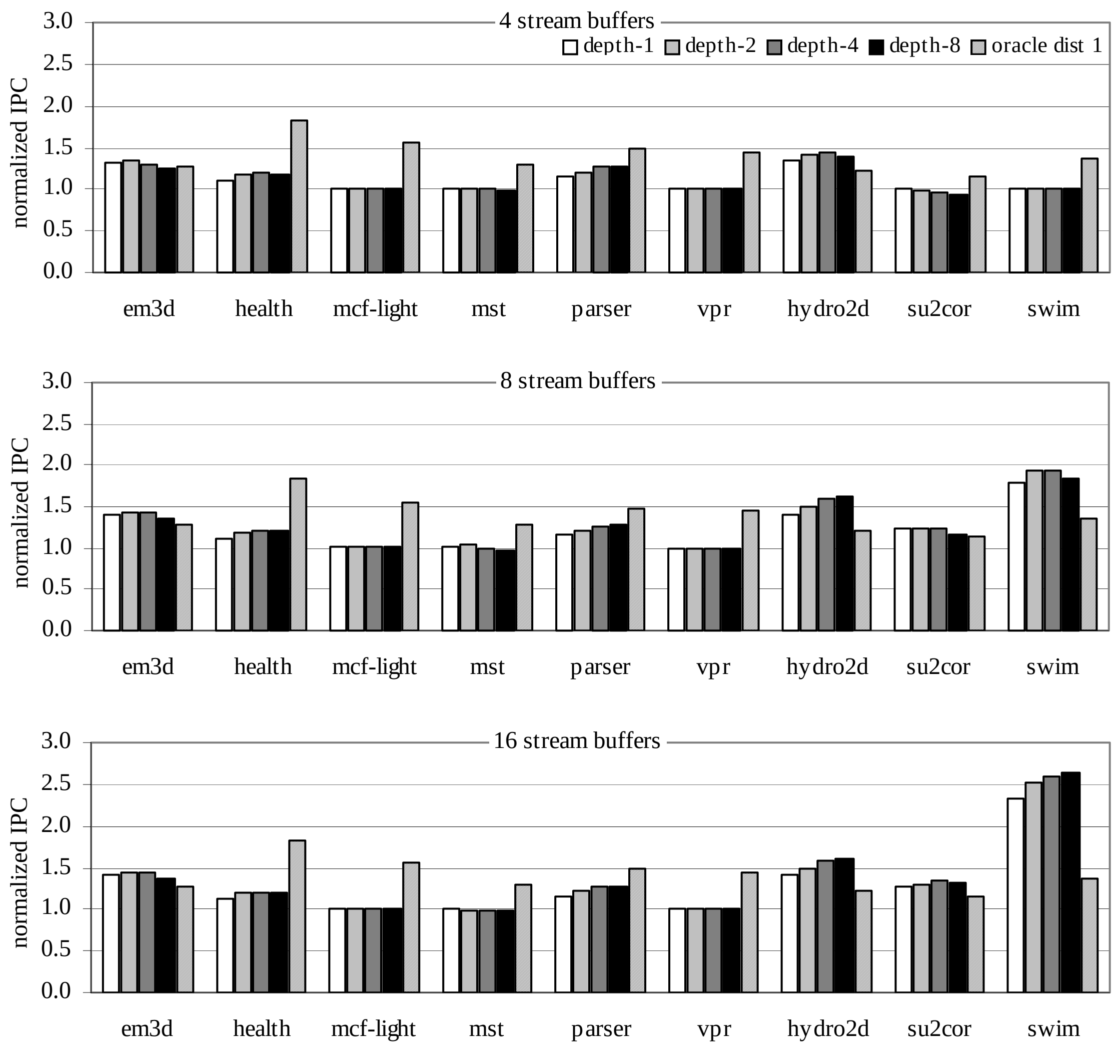


Figure 4. Stream buffer performance. This graph show the performance gain for various stream buffer depths. These graphs shows the results with 4, 8, and 16 stream buffers. The PC filter table consisted of 32 entries.

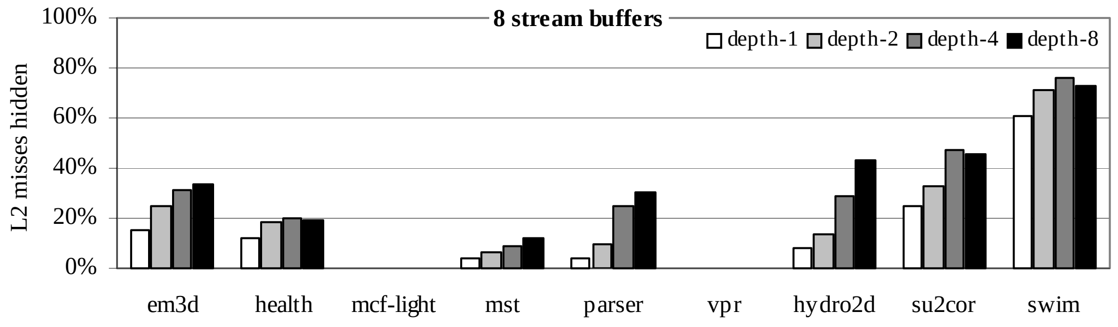


Figure 5. Hidden L2 misses. This graph shows the percentage of baseline L2 misses that were completely hidden by prefetching with 8 stream buffers with various stream depths.

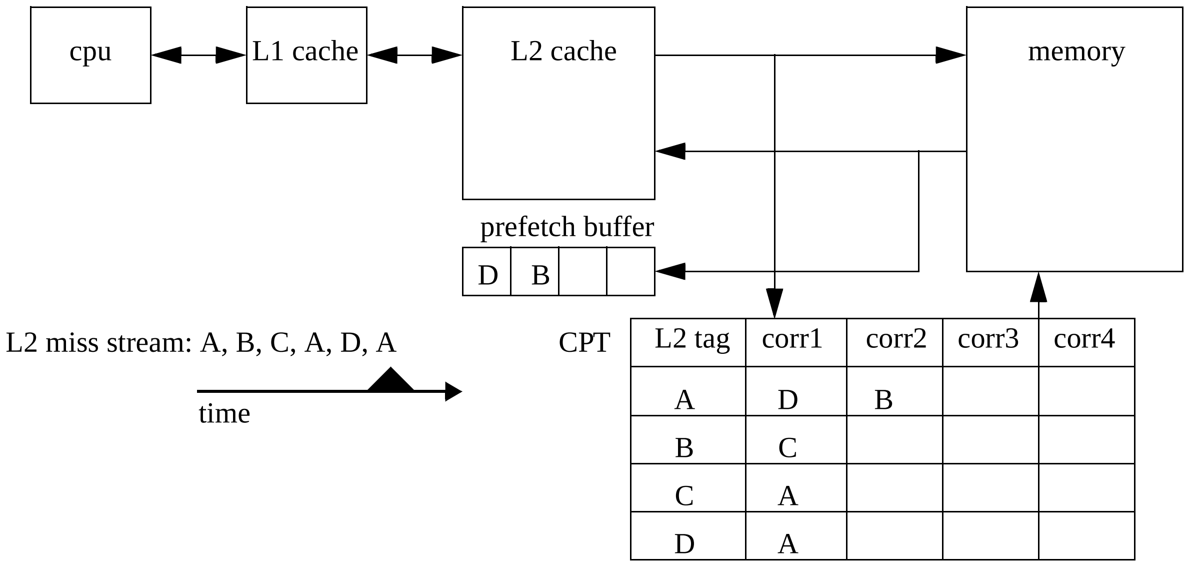


Figure 6. Markov predictor model. This figure illustrates the Markov correlation predictor that we used in our experiments. Our Markov predictor is located behind the L2 cache. Its input is the L2 cache miss stream. This CPT predicts the four unique most recently seen subsequent misses addresses associated with a L2 miss address.

distance greater than 1 because several streams can be interleaved.

Stream buffers were far from being as effective for all of the integer applications. The integer applications *em3d*, *parser,* and *health* received noticeable benefit from stream prefetching. There was hardly any improvement (or degradation) in the other integer applications. There was a very small performance gain while increasing the number of buffers from 4 to 8 and none after that since there are few streams that are accessed concurrently in these applications. Increased buffer depth was better for *parser* and worse (after 2) for *em3d*.

In summary, our experiments show that in our quest for a prefetcher at the L2 level catering to all applications, we should certainly include a stream buffer component. It is a relatively small investment in data storage hardware: For example, 16 stream buffers of 2 lines of 64 bytes represent only 2 KB which is less than 1% of the capacity of the L2 cache that we simulated; nonetheless, they yielded an average speedup of 35%. Moreover, the additional logic for comparison is not extensive: a 32-entry stream detector and the ability to compare 16 sets of two tags in parallel.

5.3 Non-strided address streams

In the previous section, we have reconfirmed that stream buffers are effective for applications that exhibited accesses to strided streams of data while having less influence on the other applications. As shown in the experiments with oracle prefetching (Figure 2), prefetching can be worthwhile for these latter cases. However, different mechanisms are required. In this section, we investigate the potential of correlation predictors as well as a predictor specifically tuned to the case of applications whose main data consists of linked data structures (LDS)..

5.3.1 Address-address (Markov) Correlation Predictor

The Markov, or address correlation, predictor was presented in [8] as a technique to prefetch at the L1-L2

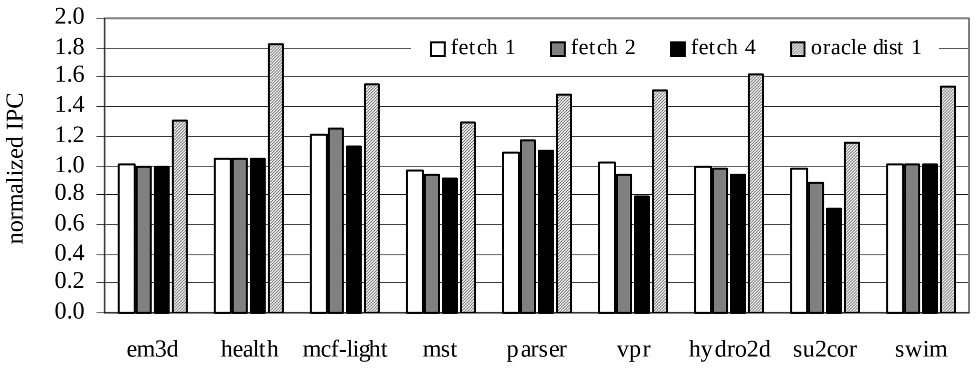


Figure 7. Markov prefetching. The graph shows the overall IPC gains from using a Markov prefetcher to prefetch L2 cache lines. Various fetch sizes, in terms of predictions, were used with each application. The correlation predictor maintains up to four predictions per CPT entry. However, depending of the fetch size, fewer than four predicted cache lines may be prefetched from memory.

boundary. We adapted the design to the L2-main memory interface as shown in Figure 6

The predictor is driven by the L2 miss stream. Its core is the Correlation Prediction Table (CPT) which, in our implementation, is organized as a fully-associative cache of *n* entries of width *m* (maximum number of cache lines correlated with an entry). Each entry in the table corresponds to a particular cache line, say A, that has missed and its "data" consists of the addresses of the lines, in most recent miss order, that missed just after A in the previous instances of misses to A. These addresses are used as predictions for prefetches whenever A misses again. Increasing *m* will increase coverage at the potential expense of bus occupancy. We introduce another parameter, the fetch size *f*, that determines the maximum number of the *m* lines that will be prefetched (note that some of the *m* entries might already be in L2 or the prefetch buffer and do not need to be prefetched). In the example of Figure 6, upon the last reference to A lines D and B would be prefetched (if not already in L2) if *f*=2, but only D would be prefetched if *f*=1 (or if *f*=2 and B is in L2)

In order to test whether address-address correlation is present in the L2 miss stream, we used generous parameters for the CPT. We simulated a Markov predictor whose CPT size is very large (*n*=32K entries; *m*=4) compared to the L2 size (4K lines of 64 bytes). To balance the trade-offs between prefetch coverage and memory bandwidth usage, we experimented with fetch sizes *f* =1, 2 and 4.

Figure 7 shows the resulting performance gain for this implementation of the Markov prefetcher. For the applications *mcf-light*, *parser*, and *health*, there was a noticeable IPC improvement over the baseline (with fetch size *f*=1) of 21%, 9%, and 5% respectively. *Mst* exhibited a slight slow-down and all other applications were practically unaffected. Increasing *f* to 2 showed a slighter larger gain for the same three applications that benefited with *f=1*, but performance degradation, probably due to the extra memory bandwidth consumed by additional inaccurate prefetches, started to occur for some of the other applications. Setting *f*=4 gave results always worse than for *f*=2.

The lackluster improvement, at the high cost of a very large CPT, can be attributed to either failure to correctly predict L2 miss addresses (poor correlations or misses in the CPT) or not predicting far enough in advance to hide the memory latency. Note that we cannot unduly restrict the size of the CPT since it must be large enough to capture the working set of the recurring missing lines. Figure 8, showing the CPT hit rate in function of its size, makes it clear that the CPT needs to be indeed very large if prefetching is to take place. For example, the hit rate for *health* is dismally low even with a CPT of 32K entries.

The Markov predictor, as presented here, intends to use a prefetch distance of one although it can

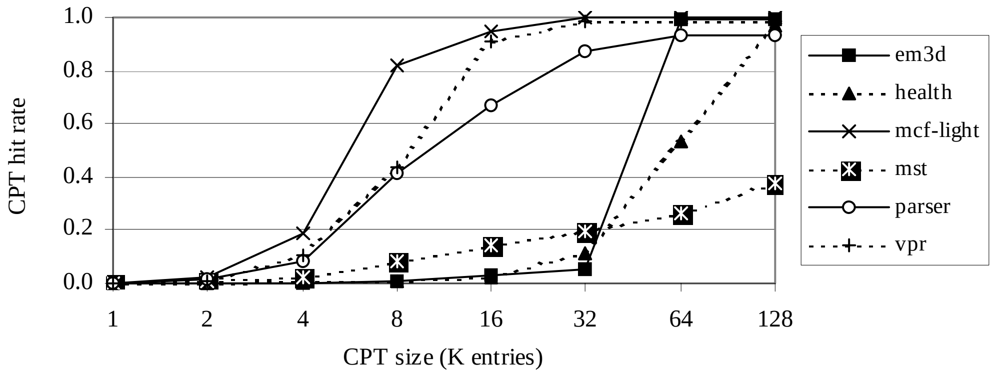


Figure 8. CPT hit rate. This graph show the hit rates for the CPT on the integer applications with an address-address Markov prefetcher. The CPT hit rate is calculated as the number of CPT lookup hits divided by the number of L2 cache misses. A CPT lookup hit is needed in order to make a prefetch.

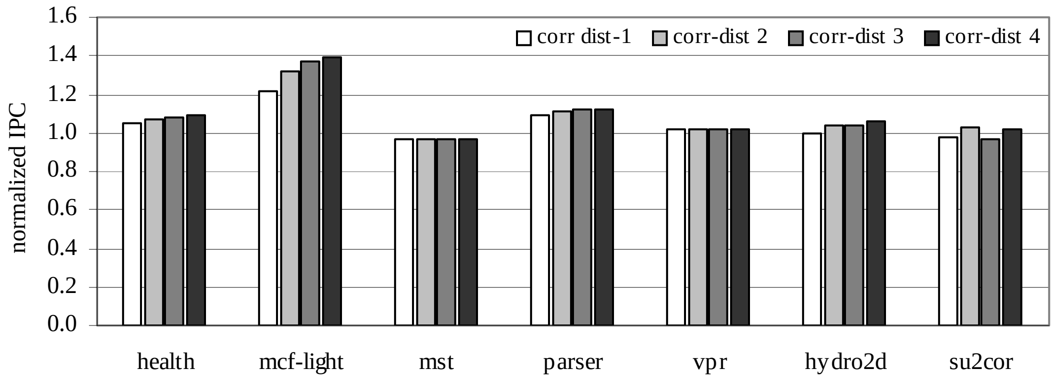


Figure 9. Increasing the prefetching distance by increasing the correlation distance. This graph shows the resulting performance from increasing the number of L2 misses between correlations built using a Markov predictor. Each correlation predicted one address (*f*=1).

bring in lines that will be useful later on. For example, for *mcf-light* (the application that benefits most from the Markov predictor) with *f*=1, only 79% of the useful prefetches had a prefetch distance of 1 while the remaining 21% were a result of the predictor's serendipity.

A means to increase the prefetch distance is to build correlations between miss addresses that occur further apart [4]. For example, correlations can be built between addresses of every other miss (correlation distance of 2). Figure 9 shows the beneficial effects of increasing the correlation distance. For example, for mcf-light when using a correlation distance of 4 and *f*=1, the IPC increased by 18% over the results with a correlation distance of 1 (and 40% over the baseline) although the miss coverage was reduced by 3%.

Overall, while the Markov predictor at the L2-main memory interface is able to be of benefit for some of the integer applications, it is fundamentally limited by the size of the CPT needed to support effectively a large set-associative L2 cache. A weakness of address-address correlation strategy is that in spite of a large L2 and CPT, it cannot predict cold misses either in the form of L2 compulsory misses or address-address pairs that have never been encountered. Furthermore, since some of the predictive power

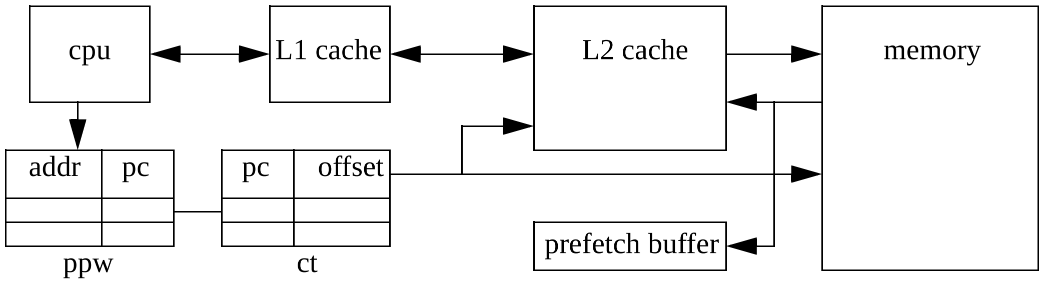


Figure 10. LDS prefetching model. The LDS prefetcher observes loads committed by the cpu in order to detect recurrent loads. Addresses predicted by the LDS prefetcher that are not in the L2 cache are prefetched to L2.

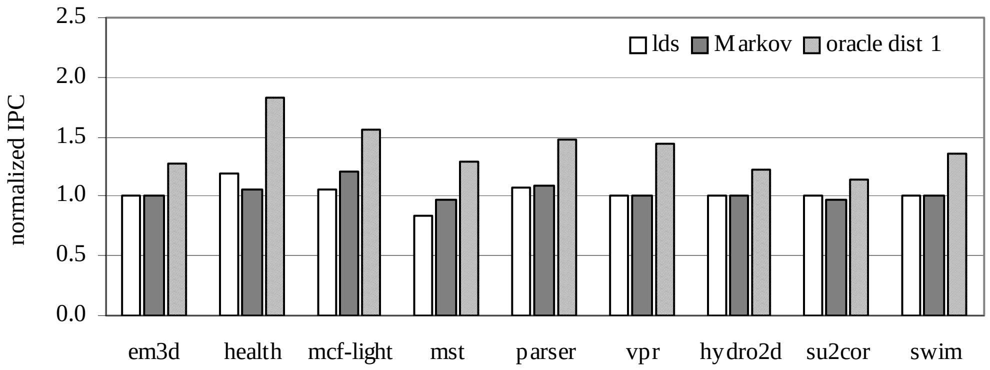


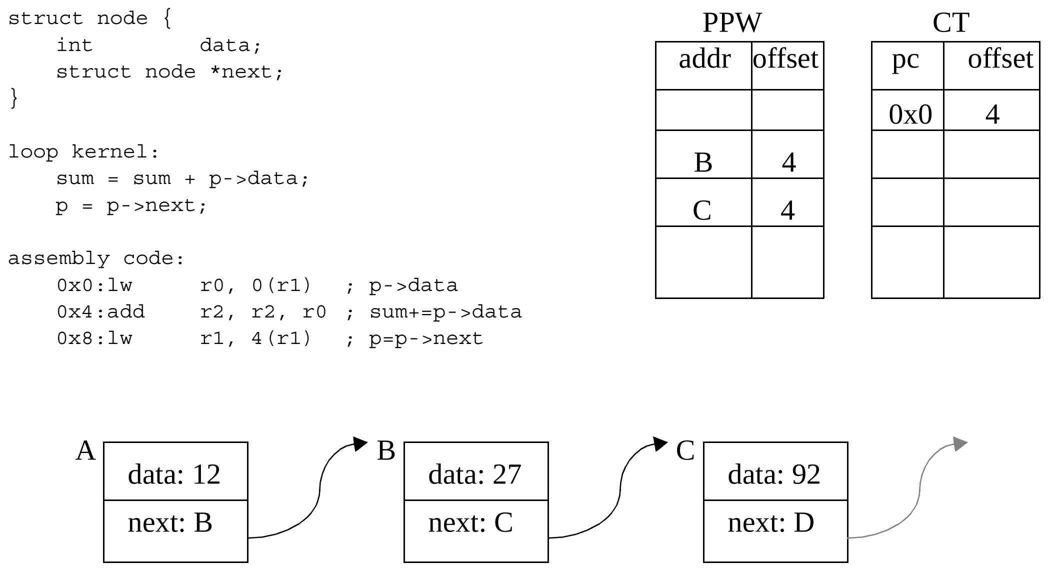
Figure 11. LDS prefetching performance. The graph shows the overall IPC gains from using LDS to prefetch linked addresses into L2. The Markov prefetcher predicted one address per correlation.

is lost because of the filtering effects of the L1 and L2 cache hits, it is not surprising that the results of this mechanism are much less interesting than those recorded for the L1-L2 interface [8]. Means to mitigate the size of the CPT either by optimizing the data part [19] or using large line granularities [1] have been investigated but, nonetheless, the method does not seem practical for prefetching for a large L2 cache with set-associative such that conflict misses are rather rare.

5.3.2 Linked data structure (LDS) predictors

Recently, a number of prefetching strategies have been proposed for the specific case of applications whose main data structures are linked lists. We examine one of them [17] that uses hardware assists to recognize links and is able to traverse links in advance of their use. We then adapt the scheme for prefetching in L2.

The basic idea in [17] is to recognize instructions that produce (define) pointer addresses, storing loaded values and their PCs in a Potential Producer Window (PPW) and, subsequently, correlate them with instructions that consume (use) the pointer addresses via a Correlation Table (CT) (Figure 10). At load commit time, if the load's base address matches an address in the PPW, a correlation between the PPW entry's PC and the committed load's value is stored in the CT. Likewise, if a load's PC is found in the CT, a prefetch will be generated for an address formed from the load's address plus the CT's entry offset value. In our experiments, we used a 128-entry PPW and a 256-entry CT as in [17] although a smaller CT, e.g.,



```
struct node {
    int         data;
    struct node *next;
}

loop kernel:
    sum = sum + p->data;
    p = p->next;

assembly code:
    0x0:lw      r0, 0(r1)   ; p->data
    0x4:add     r2, r2, r0  ; sum+=p->data
    0x8:lw      r1, 4(r1)   ; p=p->next
```

PPW

addr	offset
B	4
C	4

CT

pc	offset
0x0	4

Figure 12. LDS stream buffer detection. The figure illustrates how the original LDS algorithm was modified to stream LDS links at L2.

32-entry, yielded the same results.

It is clear that the information required for building the PPW and the CT needs to be gathered in front of the L1 cache. On a successful correlation we check whether the corresponding line is already in L2 or the prefetch buffer. If not, we prefetch into the L2 prefetch buffer as we did with other predictors. Because our experiments are geared specifically for the impact of prefetching in L2, we did not prefetch between L1 and L2 as in [17].

The performance of this LDS predictor is shown in Figure 11 (leftmost bar in each application). As could be expected, the results are far from impressive, ranging for the integer applications from a slow-down of 31% (*mst*) to a speedup of 18% (*health*) and leaving the performance of the floating-point applications unchanged. One of the problems specific with this LDS prefetcher implementation is that the prefetch distance is mostly one. Since the amount of work performed between L2 misses can be small, compared to the memory latency, not much latency hiding occurs. Thus, correct prefetches will result in only a few cycles being saved while incorrect ones will occupy the memory bus which prevents demand fetches from occurring in a more timely fashion.

In order to palliate this deficiency, we modified the LDS predictor so that it could run ahead traversing links and prefetching as necessary. In keeping with the spirit of hardware-only prefetching, we did not simulate compiler-driven prefetch engines running slices of the program [23, 24] but instead added hardware at the L2 level reminiscent of what is required for strided stream buffers..

The basic idea is to allocate a buffer (LDS buffer) to each linked data structure that is encountered. The LDS buffer will be allocated when the mechanism recognizes that an L2 miss is caused by a data structure field being accessed via a pointer. As cache lines are brought into the LDS buffer, predicted pointer fields are read from them and new lines are prefetched using the same type of throttle control as in regular strided stream buffers.

Usually, the load that will generate the first L2 miss to an individual node of a data structure will access a data field rather than the “next” field of the node as nodes are “read” before being traversed. From

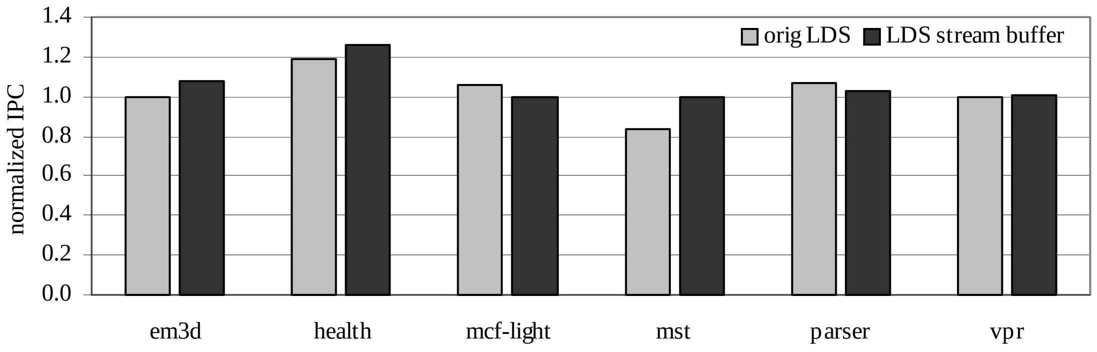


Figure 13. LDS stream buffer performance. Shown is the performance of LDS stream buffers compared with the original LDS. 16 LDS stream buffers with a depth of 4 were used.The results are IPC normalized to the base IPC.

the example in Figure 12, the load at PC 0x0 will most likely generate a miss before the one at PC 0x8. We therefore modify the PPW by storing the node's offset to the pointer with the loaded value (e.g.,. Figure 12 where the displacement "4" is stored with the addresses B and C). If a committed load's base address is found in the PPW, its PC and the PPW entry's offset value are stored in the CT (cf. Figure 12 where the entry "0x0, 4" corresponds to the load instruction at PC=0x0). Whenever an L2 miss occurs, the CT is probed for the PC of the instruction causing the miss. If this PC matches an entry in the CT, we speculate that a link has just been traversed and allocate an LDS stream buffer, using the offset to obtain the next link. The LDS stream buffer controller will use the offset field to continue streaming as prefetched lines return from memory

At allocation time, the LDS stream buffer controller requires the offset found in the CT and the full load address, not simply just the cache line tag as on an ordinary L1 miss, so that the link address can be formed. This can be implemented in a variety of ways depending on whether the L2 controller is on the same chip as L1 or not. For example, in the latter case, rather than sending the PC address and letting the L2 controller access the CT, we can send the load address, the offset from the CT, and a bit indicating whether or not there was a hit in the CT. The LDS controller, though, will always keep a copy of the CT offset value so that it can generate subsequent addresses without having to access the CT repeatedly.

As data is returned from memory to the LDS buffers, we use the same policies as for strided stream buffers: transfer into L1 and L2 when there is an LDS hit and throttle control for continuous streaming. In the cases where the "next" field of a data node does not lie on the same cache line as the base address of the node, the LDS controller will prefetch the line containing this field so that it can obtain the pointer's value in order to prefetch the next node.

We evaluated this strategy with 16 LDS stream buffers of varying maximum depths. Having more stream buffers would not be cost-performance efficient as per the data of Section 3.2. The results of our experiments are shown in Figure 13 where the IPC of the integer applications are compared under the two LDS strategies and against the baseline. As can be seen, the second LDS strategy is slightly better for *em3d* and *health* and better also for *mst* which now runs without performance degradation. The original LDS scheme works slightly better for *mcf-light* and *parser.* Overall, the second strategy is better by about 3%.

The results of LDS prefetching might seem disappointing. Some of the blame can be attributed to the solely PC-based hardware technique which limits the prefetching distance within LDS applications. More fundamentally though, some of the lack of performance improvement is caused by the short linked list traversal length in some applications. For example, in *mst*, the average length of the linked list

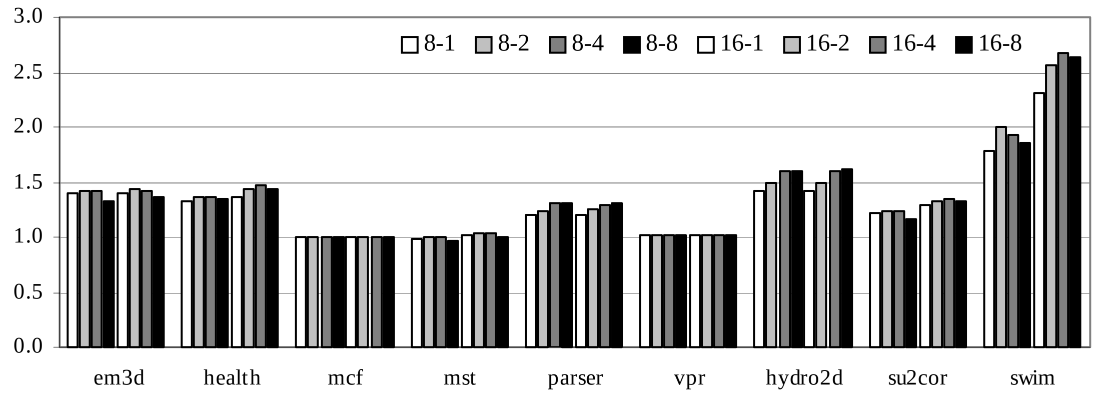


Figure 14. Hybrid predictor performance with various buffer parameters. Performance of the hybrid stream buffer prefetch with either 8 or 16 stream buffers and buffer depths of 1, 2, 4, or 8. The performance shown is normalized (to baseline) IPC.

generating a significant percentage of the L2 misses is 4 nodes and the average number of node traversed per lookup was only 1.63 nodes, thus providing little opportunity for latency hiding. Likewise, with an average of less than 2 nodes visited per list, many buffers are allocated and prefetches done unnecessarily. In contrast, *health* has much longer linked lists and the results are moderately satisfying.

In summary, LDS prefetching with LDS stream buffers accomplishes modest gains on two of the integer applications.

5.4 Hybrid predictor

Strided stream buffers have proven to be of real benefit for scientific applications while LDS stream buffers can improve the performance of some LDS applications. It seems therefore natural to build a prefetch mechanism with stream buffers with the choice of a particular type of buffer, strided or LDS, being performed dynamically.

The policy that we chose for evaluating such a hybrid predictor is to give preference to strided stream buffers since their influence on the overall performance is greatest. Upon stream buffer allocation, if the instruction generating the L2 miss can be classified as having a regular strided access, a strided stream buffer is allocated. Otherwise, an LDS stream buffer is allocated if the conditions from section 5.3.3 are fulfilled (PC matches in the CT). Because of the possibility of incorrect stream classification, we checked whether 8 buffers would be sufficient, as in the cases of the pure strided or LDS prefetchers, or whether more were needed. The results, shown in Figure 14, argue for 16 buffers of depth 2 or 4.

In Figure 15, we show the performance of the hybrid predictor compared to that of the LDS and strided stream buffers. In all cases, we used 16 buffers with depths of 2. The IPC are normalized to that of the baseline and we show also the IPC of the oracle with prefetch distance 1. On average, the hybrid predictor was able to increase the overall performance by 5% over the strided stream buffers and 14% over the LDS stream buffers. With the hybrid predictor, the scientific applications and *em3d* had IPCs superior to that of the oracle; two other applications, *health* and *parser*, showed significant performance improvements and the remaining three were essentially unaffected by the presence of the hybrid predictor.

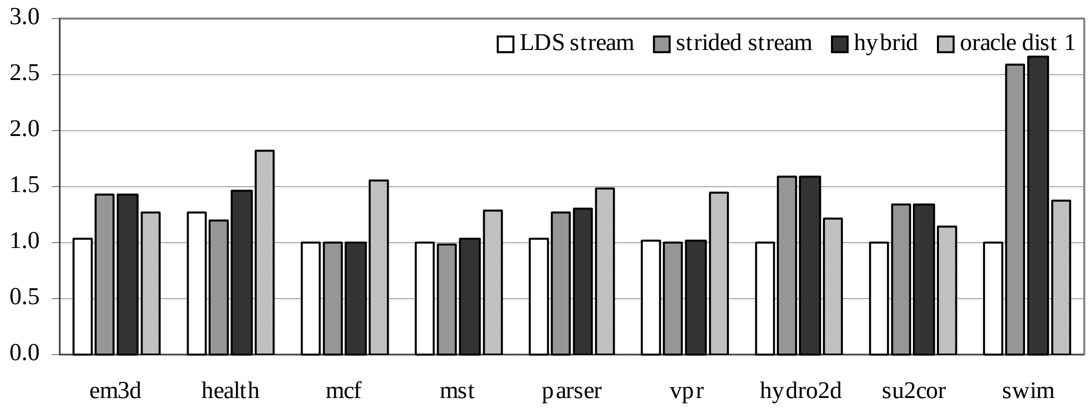


Figure 15. Performance of stream buffers. This graphs shows the performance of the three stream buffer strategies, LDS stream buffers (LDS stream), strided stream buffers (strided stream), and the hybrid prefetcher (hybrid). The IPCs were normalized to that of the baseline.

6 Summary

Hiding, or even tolerating, the long memory latencies that exist between DRAM and the cache hierarchy is a difficult challenge. In this paper, we have examined hardware-based techniques for prefetching. The benchmarks that we used for evaluation purposes include three scientific and six integer applications that have a significant number of L2 misses. Our evaluation was based on the simulation of an aggressive CPU with current memory latencies.

Our first result is that timeliness is of utmost importance. Even with perfect coverage and accuracy, oracles must prefetch more than one miss ahead in order to approach ideal configurations. We have quantified the estimated prefetch distance required for ideal speedup and found that for our benchmark suite and memory latency characteristics, it could be as large as four which can be a daunting proposition in the case of LDS applications. The redeeming good news is that few instructions at any instant generate the majority of L2 misses, therefore PC-based hardware assists can be small.

The most successful prefetchers that we examined at the L2-main memory interface are based on the concept of stream buffers. The stream buffers that we considered are allocated on L2 misses on a PC per PC basis. We confirmed previous studies that show that strided stream buffers work extremely well for scientific applications. We extended a previously published LDS prefetcher scheme so that it could be used in the context of stream buffers at the L2 level. The LDS stream buffer mechanism improved the performance of some of the LDS applications while never hindering the others. Finally, we proposed a hybrid predictor whereby stream buffers can be of either the stride or the LDS type depending on the dynamic characteristics of the application. Six of the nine applications showed significant improvements and the three others either had minuscule ones or none at all.

Successful prefetching require coverage, accuracy, and timeliness. As memory latencies increase relative to processor speeds, it is clear that timeliness is a goal that is becoming more difficult to achieve with current prefetching distances. Architects will need to design prefetching strategies with more emphasis on timeliness than in the past, possibly at the expense of coverage or accuracy. For applications with regular address access patterns, the prefetching distance can be more easily increased with almost no impact on coverage or accuracy. However, obtaining greater prefetching distances for applications with more random patterns cannot be realized uniquely by hardware prefetchers. Hybrid hardware-software techniques, including profiling and compiler directives, will be necessary. Nonetheless, increasing the prefetching distance will most likely increase mispredictions which reduce coverage and accuracy. This is not an easy trade-off to make but may be required for future designs.

References

[1] T. Alexander and G. Kedem. Distributed Prefetch-Buffer/Cache Design for High Performance Memory Systems. In *2nd International Symposium on High-Performance Computer Architecture*, pages 254–263, February 1996.

[2] D. C. Burger and T. M. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[3] M. J. Charney and T. R. Puzak. Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite. *IBM Journal of Research and Development*, 41(3):265–286, May 1997.

[4] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.

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

[6] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems,* 6(7):733–746, July 1995.

[7] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-System Design Considerations for Dynamically-Scheduled Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture,* pages 133–143, June 1997.

[8] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

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

[10]M. Karsson, F. Dahlgren, and P. Stenström. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *6th International Symposium on High-Performance Computer Architecture*, pages 206–217, January 2000.

[11]T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[12]S. Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture,* pages 24–33, April 1994..

[13]S. S. Pinter and A. Yoaz. Tango: A Hardware-Based Data Prefetching Technique for Superscalar Processors. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 214–225, December 1996.

[14]S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* pages 160–169, May 1990.

[15]J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio. Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks. U.S. Patent 4,807,110, February 1989.

[16]A. Rogers, M. Carlisle, J. Reppy, and L. Hendren.Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[17]A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[18]A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[19]T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers.In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 42–53, December 2000.

[20]A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[21]R. B. Smith, J. K. Archibald, and B. E. Nelson. Evaluating Performance of Prefetching Second Level Caches, *Performance Evaluation Review*, ACM Sigmetrics, 20(4):32–44, May 1993.

[22]G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[23]S. P. VanderWiel and D. J. Lilja. A Compiler-Assisted Data Prefetch Controller. In *Proceedings of IEEE International Conference on Computer Design*, pages 372–377, October 1999.

[24]C-L Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *International Conference on Supercomputing 2000 (ICS ‘00)*, pages 176–186, May 2000.

[25]L. Zhang, S. A. McKee, W. C. Hsieh, and J. B. Carter. Pointer-Based Prefetching Within the Impulse Adaptable Memory Controller: Initial Results. In *Proceedings of the ISCA-2000 Workshop on Solving the Memory Wall Problem*, June 2000.