Prefetching: Making the Uncommon Case Rare

Balajee Vamanan, Gwendolyn Voskuilen

ABSTRACT

Prefetch is an effective method for hiding the latency of memory accesses on cache misses. We compare two hardware prefetch methods for the L2 cache, table-based stride prefetching [1] and global history buffer based prefetching [2]. A Global History Buffer (GHB) is a FIFO that stores global miss addresses. This can be used along with an Index Table (IT) to create history of accesses that share a common property. Table-based Stride Prefetching (SPT) is simple to implement and has lower prediction latency than Global History Buffer based methods. However, GHB based methods create more opportunities for prefetch by eliminating stale data and storing history information efficiently. The increased latency due to prediction is not an issue at the lowest level of data caches where the miss penalties are high. All prefetchers are implemented in the SimpleScalar out-of-order simulator and tested using the SPEC2000 benchmark suite [3].

Our results indicate that the GHB-based prefetcher can increase IPC by up to 113%. By contrast, the stride-prefetch table achieves a maximum improvement of 35%. For integer benchmarks, SPT often performs similarly or better than GHB-based prefetch. Because SPT uses a prefetch depth of one and GHB uses a prefetch depth of 16, SPT increases overall memory traffic by an average of 13.2% while GHB increases memory traffic by 94.8%. However, GHB achieves a significantly lower L2 miss rate in comparison to SPT – 86% lower than no prefetch execution versus 45% for SPT. In addition, the GHB structure lends itself to more complex prefetching schemes that can utilize the large history information available at the GHB.

1. INTRODUCTION

Clock frequency in super-scalar processors has increased dramatically. The speed of main memory (DRAM) has been unable to keep up, both because it is slower technology and because of a need to increase DRAM density for increased memory capacity. Caches (SRAM) help to reduce the impact of this large gap in clock speed between the processor and memory. However, because SRAM is less dense than DRAM and caches must be small to facilitate quick access, caches tend to be much smaller than main memory. When large or multiple data structures do not fit in the cache, frequent cache misses result, forcing the processor to wait while memory struggles to serve data. A
method proposed for overcoming this limitation is prefetching – bringing data that has not been requested but is expected to be requested into the cache before the actual request arrives.

Hardware prefetching relies on memory access history to predict which memory locations are most likely to be requested next. Consequently, a prefetch method is characterized by a prediction algorithm that computes the subsequent addresses and data structures that store history information. In this paper, we compare the performance of multiple prediction schemes that differ both in prediction algorithms and data structures.

We focus on the lowest level data cache (in this case, L2) because modern out-of-order processors can tolerate most L1 data cache misses with relatively little performance degradation [4]. This is because the penalty associated with L1 misses is small enough to be tolerated without causing long stalls. Generating large streams of prefetches is not effective for L1 since it is small in size and associativity. Further, prefetch schemes that require multiple clocks to generate prefetch addresses are less effective at L1 than L2 because the miss latency between L1 and L2 is much less than the miss latency between L2 and main memory.

2. Stride Directed Prefetch

2.1. Background

Stride directed prefetch is a simple, fast prefetch method that is implemented in [1] for an L1 data cache. The method uses a stride prefetch table (SPT) with three fields: instruction address (PC) tag, memory address, and valid. The PC address (PC_{spt}) indexes the table and the higher-order n bits of the PC are stored as a tag. For a P bit program counter indexing a SPT with R rows, n is given by:

\[ n = P - \log_2 R \]

The memory address (MA_{spt}) field stores the last memory address accessed when the associated PC missed in the cache. The valid bit denotes that PC_{spt} and MA_{spt} are valid. On a cache miss, the PC (PC_{miss}) and memory address (MA_{miss}) associated with the miss are sent to the SPT.

If PC_{miss} matches a valid PC_{spt} in the table, there is an SPT hit. The stride is then calculated as MA_{miss} - MA_{spt}. If the stride is greater than zero, a prefetch is attempted from memory address MA_{miss} + stride. MA_{spt} is then updated with MA_{miss}. If the SPT misses, PC_{miss} and MA_{miss} are added to the table. Figure 1 shows the miss handling process.
As expected for prefetch, stride directed prefetch is most useful for large regular program structures, i.e. those with a loop that repeatedly accesses successive elements of a large data structure. Figure 2 shows an example of such a loop. In this case the stride distance between arrayA[i] and arrayA[i-1] will be constant for all i. An additional stride history field can be added to the table. Note that not only must the data structure be large enough to cause repeated cache misses as the loop executes, but it must be a structure that stores elements at consistent strides in memory.

An additional stride history field can be added to the SPT. This field is used to prevent prefetch when the previous stride does not match the current stride, as may happen when exiting a loop. Each time a prefetch is attempted, the stride is stored in the table. However, this may also have the adverse effect of reducing the number of prefetches and limiting the usefulness of the prefetcher.

```c
for (int i = 0; i < N; i++) {
  //
  //
  PX arrayA[i] = arrayB[i] + arrayC[i];
  //
  //
```

Figure 2 For Loop
2.2. IMPLEMENTATION

The stride prefetch table is implemented as a direct-mapped cache. It lies between the L2 cache and memory and is accessed on L2 cache misses coming from the data cache. Our implementation uses a prefetch table with 1K entries. In [1] this size table is shown to have similar performance to a table with infinite entries. Additionally, because the table is small and direct-mapped, access (address decode) is estimated to take only one cycle. We implemented the table with an optional stride-history field, but found the field to give little performance improvement so it was discarded.

3. GLOBAL HISTORY BUFFER

Conventional prefetchers, such as the SPT, store history information in tables which are accessed using a key or index. The tables allocate a fixed amount of history information per key, limiting the amount of information that can be stored for a given key. Further, data stored in the tables can become stale and reduce prefetch accuracy. Although performance can be improved with increased table size, implementation constraints including area and access time prohibit large tables. The Global History Buffer (GHB) approach solves these issues by decoupling history information from prefetch key matching and allowing long prefetch histories. The structure consists of two storage elements which are explained below:

1. **Global History Buffer (GHB):** The GHB is a FIFO (First In First Out) queue of L2 miss addresses. Each entry maintains a pointer (address) to the previous occurrence of the same address in the GHB. Hence, a linked list of all of previous occurrences of a given address can be extracted. Other information such as stride length between memory accesses can then be extracted from the linked list. Since GHB is a FIFO, addresses subsequent to each node in the linked list can be extracted as well.

2. **Index Table (IT):** This table handles the key matching part of the lookup. It maintains a link between the key and its most recent occurrence in GHB.

The overall structure of the global history buffer prefetcher is shown in Figure 3.
3.1. MARKOV PREFETCH

3.1.1 Background

Markov Prefetching correlates past history to predict future accesses. A history of L2 misses is maintained in data structures. When an L2 miss occurs and the miss address is present in the history, addresses that have followed the current address in the past are prefetched. The number of addresses to be prefetched depends on PREFETCH DEPTH. In addition, if the current address has appeared multiple times in the past, a stream of prefetches for each occurrence of the current address is generated. The number of streams that are generated depends on the PREFETCH WIDTH. For example, if 40, 41, 42, 43, 42, 40, 42, 43, 41, 42, 40, 43, 41, 40 is a time-ordered sequence of L2 miss addresses (block addresses), a Markov Prefetcher with a PREFETCH DEPTH of 4 and PREFETCH WIDTH of 3 would issue prefetch requests to 43, 41, 42.

The maximum number of prefetch requests a correlation prefetching method can generate is called the PREFETCH DEGREE of the method. Theoretically PREFETCH DEGREE is the product of PREFETCH WIDTH and PREFETCH DEPTH. However, it is practically limited by PREFETCH DEPTH because the size of history information that can be stored in a GHB is limited.

3.1.2 Implementation

Miss addresses are updated in the GHB in FIFO order. The L2 miss address is used as a key to index into the Index Table. When a key is not found in the IT, it is added into the table along with a pointer to the current head of the GHB. Subsequent hits to the key in the IT are used to build a linked list capturing the access pattern associated with the key in the GHB. When a key is found in the IT, the associated pointer in the IT is used to locate the...
corresponding entry (address) in the GHB. The miss address is added at the head of the GHB FIFO and this entry is
linked to the previous occurrence of the same address in the GHB by using the current pointer in the IT.

Prefetches are generated by traversing the linked list in GHB using the pointer in the IT. At every node in the
GHB, subsequent PREFETCH DEPTH entries in the GHB are prefetched provided there is enough PREFETCH
DEPTH – the number of entries between the current node and its parent node. The number of nodes traversed is
limited by PREFETCH WIDTH. Thus we get a PREFETCH DEGREE equal to the product of PREFETCH WIDTH
and PREFETCH DEPTH, provided sufficient history exists in the GHB (and miss address is sufficiently spaced out
so that we can fetch enough PREFETCH DEPTH entries).

Kyle J. Nesbit and James E. Smith determined the optimum size of both the IT and GHB to be 512 entries [2].
Each lookup in the IT (for key matching) or GHB (for linked list traversal) takes 1 clock of latency. Hence prefetch
address generation would take a minimum of 2 clocks (for the first node in the linked list) and an extra clock for
each additional node traversed (limited by PREFETCH WIDTH). This latency is accounted in our performance
evaluation. This scheme is referred as G_AC in the results section of the document.

3.2. DISTANCE PREFETCH

3.2.1 Background

Distance prefetching is a generalization of Markov prefetching [2]. Distance prefetching uses the distance
between two consecutive global miss addresses, called an address delta, to index the history information. The
prefetch address is formed by adding the address delta to the current miss address. For illustration, let us assume 40,
41, 42, 43, 42, 40, 42, 43, 41, 42, 40, 43, 41, 40 to be a time-ordered sequence of L2 miss addresses (block
addresses). The address delta stream can be computed as 1, 1, 1, -1, -2, 2, 1, -2, 1, -2, 3, -2, -1. A Distance prefetcher
with a PREFETCH DEPTH of 4 and PREFETCH WIDTH of 3 would prefetch addresses 38(40 – 2), 42(40 + 2),
41(40 + 1). Thus Distance prefetching can prefetch some of the addresses that Markov prefetching can, and it can
prefetch other delta streams that occur in a global miss address stream [2].

3.2.2 Implementation

A Global History Buffer can be used to implement Distance prefetching by using the delta address instead of the
actual address to index into the Index table. The GHB stores the global miss address, as for Markov, meaning that
each time an IT hit occurs, the address deltas between subsequent entries in the GHB must be calculated.
Nesbit and Smith found the optimum size of both the IT and GHB to be 512 entries [2]. Each lookup in the IT (for key matching) or GHB (for linked list traversal) takes a clock of latency. Hence prefetch address generation would take a minimum of 2 clocks (for the first node in the linked list) and an extra clock for each additional node traversed (limited by PREFETCH WIDTH). This latency is accounted in our performance evaluation. This scheme is referred as G_DC in the results section of the document.

3.2.3 Filtering invalid entries in the GHB and IT

The GHB is a circular buffer and hence the pointer wraps around once each time the number of global misses equals the size of the GHB. Existing entries in the GHB are overwritten, corrupting the linkage between the IT and GHB. Further, entries are overwritten which are linked inside the GHB, corrupting the linked list of history information in GHB. For this reason, a prefetch scheme using a GHB can mispredict because of invalid pointers. This cannot be completely avoided because of the limited size of GHB, but the frequency of this case can be reduced by storing wider pointers in IT and GHB. With wider pointers, whenever the difference between the pointer in IT and current head of GHB FIFO is greater than the size of GHB, further traversal into GHB is not done and prefetch is aborted. Also, whenever the difference between the GHB entry and its pointer to the next entry is greater than the size of the GHB, the prefetch is aborted. Unfortunately, even this scheme can cause mispredictions because the size of the pointers that can be stored in GHB and IT is not infinite. However, Nesbit and Smith in [2] found that increasing the width of these pointers by four bits makes such incorrect predictions highly improbable. We have summarized the effectiveness of this filtering in the results section of the document.

3.2.4 Absolute Prefetch Distance

We introduce a subtle variation to Distance Prefetching [1]. Instead of using the delta address as a key into the index table, we use the absolute delta of the miss address stream. Since this can potentially reduce the size the size of Index table by half, we wanted to compare the performance of this scheme with the Distance Prefetching. This scheme is referred as ABS MODE in the results section of the document where the performance of this scheme is compared and summarized.

4. RESULTS

4.1. Simulation Methodology

The Global History Buffer based Markov and Distance prefetchers and table-based stride prefetcher were implemented in the SimpleScalar out of order simulator using two sets of parameters as shown in Table 1. The
default parameters are meant to be more realistic and the paper parameters are the ones used in [2]. Four variations of the distance prefetcher (G_DC) were used. We call these Normal-No Filter (Norm, NF), Normal-Filter (Norm, F), Absolute-No Filter (Abs, NF), and Absolute-Filter (Abs, F) (see Section 3.2.2). The Markov prefetcher (G_AC) was tested with both Filter and No-Filter parameters. In total eight prefetch variations (including no prefetch) are tested.

Table 1 Sim-outorder Configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Parameters</th>
<th>Paper Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache Size/Configuration</td>
<td>Split, 32KB 2-way set associative, 32-byte block</td>
<td>Split, 16KB 2-way set associative, 32-byte block</td>
</tr>
<tr>
<td>L1 Cache Latency</td>
<td>2 cycles</td>
<td>2 cycles</td>
</tr>
<tr>
<td>L2 Cache Size/Configuration</td>
<td>Unified, 2MB 8-way set associative, 128-byte block</td>
<td>Unified, 512KB 2-way set associative, 128-byte block</td>
</tr>
<tr>
<td>L2 Cache Latency</td>
<td>12 cycles</td>
<td>12 cycles</td>
</tr>
<tr>
<td>Memory Size</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
<tr>
<td>Memory Latency</td>
<td>300 cycle first chunk/30 cycle inter-chunk</td>
<td>300 cycle first chunk/30 cycle inter-chunk</td>
</tr>
<tr>
<td>Memory Queue Depth</td>
<td>Infinite</td>
<td>Infinite</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>16 bytes</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Translation Lookaside Buffer Size/Configuration</td>
<td>256 entries, fully associative</td>
<td>256 entries, fully associative</td>
</tr>
<tr>
<td>Translation Lookaside Buffer Latency</td>
<td>300 cycle miss</td>
<td>300 cycle miss</td>
</tr>
<tr>
<td>Branch Misprediction Penalty</td>
<td>15 cycles</td>
<td>15 cycles</td>
</tr>
<tr>
<td>Branch Prediction Table Size</td>
<td>8K entries</td>
<td>8K entries</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>4 entries wide</td>
<td>4 entries wide</td>
</tr>
<tr>
<td>Reorder Buffer Size</td>
<td>128 entries</td>
<td>128 entries</td>
</tr>
<tr>
<td>Load Store Queue Size</td>
<td>64 entries</td>
<td>64 entries</td>
</tr>
<tr>
<td>Return Address Stack</td>
<td>32 entries</td>
<td>32 entries</td>
</tr>
</tbody>
</table>

The SPEC2000 benchmark suite was used to test each of the prefetchers. The benchmarks used are listed in Table 2. The sixtrack and lucas benchmarks were excluded because they generated unsupported system calls preventing them from running to completion. The first billion instructions of the benchmarks were skipped and the following billion were simulated.

Table 2 SPEC2000 Benchmarks

<table>
<thead>
<tr>
<th>Integer Benchmarks</th>
<th>Floating Point Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>ammp</td>
</tr>
<tr>
<td>crafty</td>
<td>applu</td>
</tr>
<tr>
<td>eon</td>
<td>mgrid</td>
</tr>
<tr>
<td>galgel</td>
<td>asi</td>
</tr>
<tr>
<td>twolf</td>
<td>swim</td>
</tr>
<tr>
<td>gap</td>
<td>equake</td>
</tr>
<tr>
<td>gcc</td>
<td>facerec</td>
</tr>
<tr>
<td>gzip</td>
<td>fma3d</td>
</tr>
</tbody>
</table>

The SPEC2000 benchmark suite was used to test each of the prefetchers. The benchmarks used are listed in Table 2. The sixtrack and lucas benchmarks were excluded because they generated unsupported system calls preventing them from running to completion. The first billion instructions of the benchmarks were skipped and the following billion were simulated.
We modified the cache access function in SimpleScalar to allow us to easily pass information regarding the PC and the origin (data vs. instruction cache) of an L2 miss. This is not realistic for a hardware implementation, as passing information needed for the uncommon case of an L2 miss through the data hierarchy will increase the cache logic. However, because of the structure of the SimpleScalar simulator, to ensure that a prefetch decision is made and executed \textit{as soon as} an L2 miss is detected, and not when the L2 miss returns, the prefetcher must be implemented as part of the cache hierarchy. This requires the cache structures to have knowledge of instruction addresses for indexing into the SPT and IT. Additionally, because we target data cache misses, we track the origin of misses in the unified L2 cache. We only prefetch when an L2 miss comes from the data cache. Finally, the caches hold an extra state value denoting whether a cache block was prefetched or not. This state allows us to check whether a prefetched block is subsequently used. The ratio of prefetch hits to overall prefetches is particularly important since prefetched blocks are placed directly in the L2 cache. A low prefetch hit rate indicates that the prefetcher may be replacing useful data in addition to not prefetching needed data.

4.2. \textbf{Performance Comparison}

\textit{4.2.1 Comparison of GHB Based Schemes}

We compare the six versions of the GHB prefetcher with regard to IPC. Figure 4 shows the results for each benchmark using the default configuration. It is clear that the G_DC(Norm) prefetchers outperform both the GDC(Abs) and G_AC prefetchers in a majority of benchmarks. The only exceptions to this occur for \textit{art} and \textit{perlbmk} where G_AC shows a slight improvement over the others. However, because G_AC shows improvement only for these two, and in the other cases performs poorly, we consider the G_DC(Norm) prefetchers to be the best option of all of these in terms of increasing IPC. The results using the paper configuration are similar and can be found in the Appendix.
Figure 4 Global History Buffer Prefetchers - IPC (Default Configuration)

Both the filter and no-filter versions of G_DC(Norm) show similar IPC. However, from the memory traffic comparison in Figure 5 it is evident that the filtered version of the prefetcher considerably reduces memory traffic (see Appendix for paper configuration). This is because the filtered version prevents prefetches from occurring after the global history buffer has begun overwriting its entries, causing incorrect links between the entries. We therefore take the G_DC(Norm, F) to be the best of the GHB schemes and use it for further analysis with the SPT and No-Prefetch cases.
The intention of prefetch is to increase IPC and reduce the L2 miss rate. Both G_DC and SPT achieved positive improvement in IPC over no prefetching. For ammp, mgrid, and swim with the default simulator configuration, G_DC showed over 100% improvement in IPC. SPT achieved a lower improvement, with the best IPC at about 35% improvement for the mgrid and mcf benchmarks (paper configuration is similar, see appendix). This indicates that both G_DC and SPT have a performance advantage over no prefetching, although SPT’s advantage tends to be smaller. G_DC’s advantage is primarily due to more aggressive prefetching – up to a degree of 16 versus a degree of 1. However it is also important to note that for a number of benchmarks G_DC achieves a negative improvement in IPC – art and perlbench for the default configuration, and crafty, gcc, twolf, and vortex for the paper configuration. In all of these cases, SPT outperforms G_DC. With the exception of art, these benchmarks are integer benchmarks, indicating that SPT may be more reliable for integer programs. This is also likely related to prefetch depth. G_DC assumes the program will need to access multiple data blocks at regular intervals, a case more common in floating point programs. Integer programs tend to have a less predictable data access pattern and because SPT prefetching is more conservative, it is less likely to prefetch unneeded cache blocks.

**Figure 5 Percentage Increase in Memory Traffic (Default Configuration)**

### 4.2.2 Overall Performance Comparison: SPT and GHB

The intention of prefetch is to increase IPC and reduce the L2 miss rate. Both G_DC and SPT achieved positive improvement in IPC over no prefetching. For ammp, mgrid, and swim with the default simulator configuration, G_DC showed over 100% improvement in IPC. SPT achieved a lower improvement, with the best IPC at about 35% improvement for the mgrid and mcf benchmarks (paper configuration is similar, see appendix). This indicates that both G_DC and SPT have a performance advantage over no prefetching, although SPT’s advantage tends to be smaller. G_DC’s advantage is primarily due to more aggressive prefetching – up to a degree of 16 versus a degree of 1. However it is also important to note that for a number of benchmarks G_DC achieves a negative improvement in IPC – art and perlbench for the default configuration, and crafty, gcc, twolf, and vortex for the paper configuration. In all of these cases, SPT outperforms G_DC. With the exception of art, these benchmarks are integer benchmarks, indicating that SPT may be more reliable for integer programs. This is also likely related to prefetch depth. G_DC assumes the program will need to access multiple data blocks at regular intervals, a case more common in floating point programs. Integer programs tend to have a less predictable data access pattern and because SPT prefetching is more conservative, it is less likely to prefetch unneeded cache blocks.
Both SPT and G_DC reduce miss rate (Figure 7) for floating point benchmarks. For a number of the integer benchmarks, in particular *crafty, eon, gcc, gzip, perlbmk, twolf*, and *vortex*, the miss rate does not decrease, indicating that prefetch is not having the desired effect in the L2 cache. However, because percent improvement in IPC for these cases is not negative, we conclude that while prefetch is not improving performance for many of the integer benchmarks, it is also not hurting performance. In some cases, IPC improves even though miss rate does not go down with the addition of prefetching. The notable exception is for *perlbmk* which has a performance degradation in terms of IPC and does not show improvement in L2 miss rate. Because only one benchmark out of 24 showed IPC degradation, we argue that in general SPT and GHB improve performance. From the graphs it is clear that G_DC has a much larger performance improvement over SPT. This is most apparent for the *anmp* benchmark where the miss rate drops from 0.8misses/access to 0.1misses/access for the G_DC. Simultaneously, IPC improves by 110%.

**Figure 6 Percent Improvement in IPC Over No-Prefetch Execution (Default Configuration)**
Figure 7 L2 Miss Rate (Default Configuration)

One notable performance advantage of SPT over G_DC is the relatively lower memory traffic. Partially because of prefetch depth and also because, as explained in following sections, SPT has a higher hit rate than G_DC, SPT results in much less memory traffic between the L2 cache and main memory. Traffic is defined as the sum of L2 cache misses (not including prefetches that miss in L2 cache) and prefetches that require an access to main memory to bring a block into the cache. Figure 8 shows the relative increase in memory accesses for SPT and G_DC over the no-prefetch case. For every benchmark, SPT shows a lower increase than G_DC. The range of SPT increases is 0–55%. For G_DC this range is 0–862%. The average SPT increase is 13.2% compared to 94.8% for the G_DC prefetcher. Removing the outliers from the G_DC range (art for the default configuration and gcc for the paper configuration, see Appendix), the G_DC range becomes 0 – 281%. G_DC therefore has a much more dramatic effect on memory accesses. Whether this effect will lead to degraded performance of memory for demand accesses depends on the ability of memory to simultaneously handle multiple accesses. Memory must have deep request queues and total available memory bandwidth must be high.
It is clear that by most performance metrics, G\_DC outperforms SPT. Some exceptions exist, but even where G\_DC performs worse than SPT, overall performance, as measured by IPC is not degraded.

### 4.2.3 Prefetch Effect on L2 Cache

As discussed above, both SPT and G\_DC reduce the L2 miss rate. Looking at the reduction in L2 misses in the presence of prefetch for the paper configuration (Figure 9) it is immediately apparent that for a number of benchmarks G\_DC actually increases the L2 miss rate. Further for half of the integer benchmarks, SPT outperforms G\_DC by a margin of up to 50 percentage points. By contrast, the default configuration shows that G\_DC outperforms SPT more consistently. The reason for the difference between the paper and default results for the integer programs is due to cache sizes. Note that the performance of art with G\_DC prefetch varies a lot between the paper and default configuration. We assume this is because the art benchmark is especially sensitive to cache parameters. The gcc benchmark is similar in variance. Overall, G\_DC reduces L2 cache misses by 65-75% for floating point benchmarks. SPT on the other hand shows an improvement of roughly 30% for the same benchmarks.

For integer benchmarks, the difference in misses is smaller and both have average improvements in the 10-20% range.
Figure 9 Percent Reduction in L2 Misses Due to Prefetching (Paper Configuration)

Figure 10 Percent Reduction in L2 Misses Due to Prefetching (Default Configuration)
On a prefetch attempt, the SPT prefetcher prefetches only the address as \( M_{\text{spt}} \) + stride, for a prefetch depth of one. The G_DC prefetcher prefetches up to 16 addresses for a given L2 miss. Figure 11 indicates that for the G_DC prefetcher, the average prefetch degree is larger for floating point applications than for integer applications. By contrast, SPT maintains a relatively consistent prefetch degree across floating point and integer applications. The maximum prefetch degree for G_DC is 7.4, slightly less than half of the allowable prefetch degree. For SPT the maximum prefetch degree is 0.8, over half the allowable prefetch degree. This proves that the average prefetch degree is always less than the theoretical prefetch degree (16 for G_DC). Further, the maximum prefetch degree must be much greater than the desired prefetch degree, from these results we estimate 2x greater.

**Figure 11 Number of Prefetches Attempted per L2 Cache Miss**

### 4.2.4 Effectiveness of Prefetch

Prefetch effectiveness, measured by the proportion of prefetch blocks that subsequently result in a cache hit, is shown in Figure 12. SPT shows a higher prefetch hit rate than G_DC for all benchmarks. In multiple cases SPT comes close to reaching a rate of a hit for every prefetch. Because G_DC has better performance in terms of L2 miss rate reduction and IPC, it is unlikely that this rate stems from an overall performance advantage. Instead, because G_DC is prefetching multiple addresses where SPT prefetches only one, G_DC is fetching more unused cache
blocks, leading to a lower hit rate. Even with the lower hit rate, G_DC achieves higher overall performance. We predict that for a prefetch degree of 16 for the SPT, the SPT hit ratio will be worse than the G_DC hit ratio since the GHB holds a more complete history.

Figure 12 Prefetch Hits per Attempted Prefetch (Default Configuration)

Finally, we consider the effect of filtering on prefetch hit rate. Figure 13 shows the prefetch hit rate for the G_DC(Norm, F) and G_DC(Norm, NF) prefetchers. It is apparent that there is very little advantage of the filtered version of the prefetcher over the non-filtered version as far as improving the hit ratio. Instead the primary advantage of the filtered G_DC prefetcher is in its smaller effect on memory traffic.
Figure 13 Hit Ratio for Filtered vs. Not Filtered G_DC Prefetcher (Default Configuration)

Figure 14 Percent Increase in Memory Traffic Over No-Prefetch Execution (Default Configuration)
5. Conclusion

Global History Buffer based prefetching methods perform better than table-based Stride Directed prefetching across most benchmarks. The average increase in IPC of GHB over no prefetching is 25%. For SPT the average improvement over no prefetching is 8%. One drawback of GHB-based prefetching is a larger increase in average total memory traffic. Our results indicate that the GHB-based prefetchers increase memory traffic over no-prefetching by 94.8%. The SPT increases memory traffic over no-prefetching by 13.2%. These are very similar to results obtained in [2]. Using the cache parameters presented in [2] did not have a substantial effect on the patterns of performance between benchmarks for different metrics. Average IPC for example varied 1% between the paper and default cache configurations.

We conclude that GHB methods are better than SPT because they have a larger prefetch degree and can capture non-unit stride lengths in the GHB. Even if the prefetch degree of SPT were to increase, the SPT would be limited to prefetching addresses at an offset equal to some multiple of the stride.

GHB based methods are better because they achieve a higher overall performance. They increase IPC while decreasing the L2 cache miss rate. Prediction accuracy is less important than the performance metrics of IPC and L2 cache miss rate.

1. GHB based methods inherently eliminate the stale data and therefore increase prediction accuracy for higher prefetch degrees.

2. By preserving the cache miss history information, GHB offers opportunities to implement complex prediction algorithms not possible with a simple table.

3. GHB achieves better performance than the SPT with a smaller table.

Increased prediction latency arising from the need to walk through the GHB table does not impact the performance of prediction schemes at the lowest level of cache hierarchy where miss penalty is very crucial.

An extension of this analysis would be to compare Absolute Distance Prefetching with Distance Prefetching by halving the size of the Index Table and GHB. Other extensions include integrating GHB with other complex prediction schemes, storing history information in other data structures such as a tree or hash to improve the efficiency of history storage, and implementing a stream buffer for prefetchers with large prefetch degree.

6. References


[3]. *SimpleScalar Simulator*. s.l.: SimpleScalar LLC.

7. Appendix

Figure 15 Global History Buffer Prefetchers - IPC (Paper Configuration)
Figure 16 Percent Increase in Memory Traffic (Paper Configuration)

Figure 17 IPC (Paper Configuration)

Figure 18 IPC (Default Configuration)
Figure 19 Percent Improvement in IPC Over No Prefetch Execution (Paper Configuration)

Figure 20 Miss Rate for L2 Cache (Paper Configuration)
Figure 21 Number of Prefetches Attempted Per L2 Cache Miss (Paper Configuration)

Figure 22 Prefetch Hit Ratio (Paper Configuration)
Prefetch Hit Ratio

- G_DC (Norm, NF)
- G_DC (Norm, F)

Floating Point Benchmarks:
- ammp
- applu
- apj1
- art
- enuake
- fastrec
- fma.ad
- galois
- mesa
- mgrid
- swim
- wupwpr
- xjpeg
- crafty
- eon

Integer Benchmarks:
- gap
- gcc
- gzip
- mcf
- parser
- perlbench
- twolf
- vortex
- vpr