Data Cache Prefetching Using a Global History Buffer

Kyle J. Nesbit and James E. Smith
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
{nesbit, jes}@ece.wisc.edu

Abstract

A new structure for implementing data cache prefetching is proposed and analyzed via simulation. The structure is based on a Global History Buffer that holds the most recent miss addresses in FIFO order. Linked lists within this global history buffer connect addresses that have some common property, e.g. they were all generated by the same load instruction. The Global History Buffer can be used for implementing a number of previously proposed prefetch methods, as well as new ones.

Prefetching with the Global History Buffer has two significant advantages over conventional table prefetching methods. First, the use of a FIFO history buffer can improve the accuracy of correlation prefetching by eliminating stale data from the table. Second, the Global History Buffer contains a more complete (and intact) picture of cache miss history, creating opportunities to design more effective prefetching methods. Global History Buffer prefetching can increase correlation prefetching performance by 20% and cut its memory traffic by 90%. Furthermore, the Global History Buffer can make correlations within a load's address stream, which can increase stride prefetching performance by 6%. Collectively, the Global History Buffer prefetching methods perform as well or better than the conventional prefetching methods studied on 14 of 15 benchmarks.

1. Introduction

Throughout the development of microprocessors, trends in both underlying semiconductor technology and in microarchitecture have significantly reduced processor clock periods. Meanwhile, the major trend in main memory technology has been in the direction of higher densities with memory access times decreasing much less than processor cycle times. These trends have significantly increased main memory latencies when measured in processor clock cycles.

To avoid large performance losses due to long memory access delays, microprocessors rely on a hierarchy of cache memories. Unfortunately, cache memories are not always effective due to limited cache capacity (and to a lesser extent, limited associativity). To at least partially overcome the limitations of cache memories, data can be prefetched into the cache.

The simplest prefetch methods are sequential, that is they access cache lines that immediately follow the current cache line [8,12,13,15]. Early sequential methods always prefetch after each cache miss [13], while more recent sequential methods [15] wait to issue prefetches until a sequential access pattern is detected. Once sequential prefetches are issued and turn out to be correct, the *degree* of the prefetching is increased until the prefetch can completely hide the latency of a miss to main memory. *Prefetch degree* is the maximum number of cache lines prefetched in response to a single prefetch request. For longer memory latencies, a higher degree is required in order for prefetched data to be returned in time to avoid a cache miss.

More advanced prefetch methods use tables to record history information related to data accesses (Figure 1a). A prefetch table may contain stride information, or information describing more complex access patterns, e.g. as in Markov prefetching [6] (specifics are described below). The table is accessed with a key, e.g. the program counter of a load instruction or a miss address. Then, history information read from the table is used for predicting memory lines to be prefetched.

Although they are simple, conventional table-based methods [4,6,7,9,10,11,12,14] are relatively inefficient, in that they reserve a fixed amount of history space per prefetch key. In some cases, data in some entries may sit in the table for a very long time and become stale (i.e. it represents history in the distant past which no longer reflects current conditions). This leads to ineffective prefetches when the table entry is eventually accessed.

We propose an alternative structure for holding prefetch history (Figure 1b). All address history is held in a FIFO table, the *Global History Buffer (GHB)*, with all global miss addresses being placed in the table at

the bottom and removed from the top. GHB history information is maintained in linked lists, which are accessed indirectly via a hash table. This method not only reduces stale history data, but it also allows a more accurate re-construction of the history of access patterns, and therefore leads to more effective prefetching algorithms. As we will show, the proposed indirect method supports improved stride and correlation prefetching algorithms.

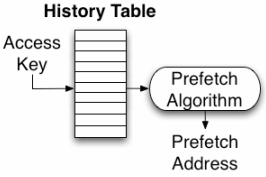

Figure 1a: Basic Prefetch Table

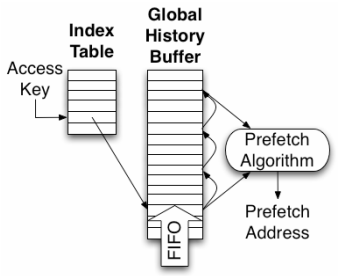

Figure 1b: Global History Buffer Prefetch Structure

As with most recent research on data prefetching [6,7,10,11,12], we focus on the lowest level data cache (in our case the L2) because modern out-of-order processors can tolerate most L1 data cache misses with relatively little performance degradation. The GHB methods can be extended to an L3 cache, if present.

2. Table-Based Prefetching

As mentioned above, we are interested in data prefetch methods that prefetch based on history information recorded in a table. The conventional implementations of these methods fit the general structure of Figure 1a.

2.1. Stride Prefetching

Conventional Stride Prefetching [5] uses a table (Figure 2) to store stride-related local history information. The *program counter* (PC) of a load instruction indexes the table. Each table entry holds the load's most recent stride (the difference between the two most recently preceding load addresses), last address (to allow computation of the next local stride), and state information describing the stability of the load's recent stride behavior. When a prefetch is triggered, addresses *a+s, a+2s, . . ., a+ds* are prefetched -- where *a* is the load's current target address, *s* is the detected stride and *d* is the prefetch degree, an implementation dependent prefetch look-ahead distance; more aggressive prefetch implementations will use a higher value for *d*. Originally, Chen and Baer [5] used a look-ahead PC (LA-PC) to prefetch ahead, although we found that using a prefetch look-ahead distance reduces complexity and is more effective.

When originally proposed, this method was applied to a single L1 cache, and all load PCs were applied to the Stride Prefetching table. In a hierarchical cache scheme, using all load PCs results in relatively high demand on L1 and L2 cache ports. Moreover, as noted earlier, L1 prefetching provides relatively little benefit beyond L2 prefetching. Hence, we implement Stride Prefetching into the L2 cache only, i.e. by using only the PCs and addresses of the loads that miss in the L2 cache.

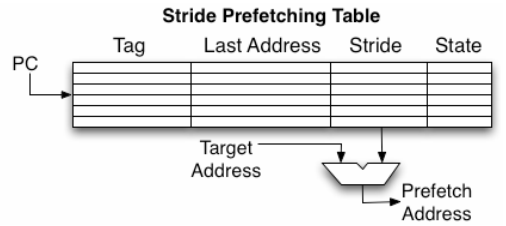

Figure 2: Arbitrary Stride Prefetching Table

2.2. Markov Prefetching

Markov Prefetching [6] is an example of a correlation prefetching method. Correlation prefetching uses a history table to record consecutive address pairs. When a cache miss occurs, the miss address indexes the correlation table, Figure 3. Each entry in the Markov correlation table holds a list of addresses that have immediately followed the current miss address in the past. When a table entry is accessed, the member(s) of its address list are prefetched, with the most recent miss address first. The left side of Figure 3 illustrates the state of the correlation table after processing the miss address stream shown at the top of the figure.

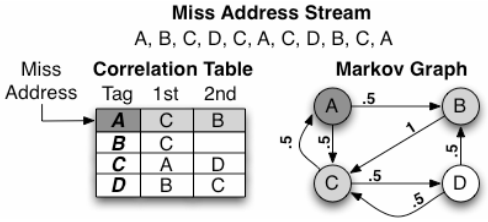

Figure 3: Markov Prefetching

Markov prefetching models the miss address stream as a Markov graph – informally, a probabilistic state machine. Each node in the Markov graph is an address and the arcs between nodes are labeled with the probabilities that the arc's source node address will be immediately followed by the target node address. Each entry in the correlation table represents a node in an associated Markov graph, and its list of memory addresses represents arcs with the highest probabilities.

Hence, the table maintains only a very crude approximation to the actual Markov probabilities. The right side of Figure 3 is the Markov transition graph that corresponds to the example miss address stream.

2.3. Distance Prefetching

Distance Prefetching [9] is a generalization of Markov Prefetching. Originally, Distance Prefetching was proposed for prefetching TLB entries, but the method is easily adapted to prefetching cache lines. In this adaptation, Distance Prefetching uses the distance between two consecutive global miss addresses, an *address delta*, to index the correlation table. Each correlation table entry holds a list of deltas that have followed the entry's delta in the past. Figure 4 illustrates the address delta stream for the miss address stream in the previous example (Figure 3), and the state of the correlation table after processing the delta stream. Distance Prefetching is considered a generalization of Markov Prefetching because one delta correlation can represent many miss address correlations. By generalizing Markov Prefetching, Distance Prefetching is capable of prefetching most of the reference patterns that Markov Prefetching can. Plus, with the data it has available, it can also detect and prefetch delta access patterns that occur in the global miss address stream.

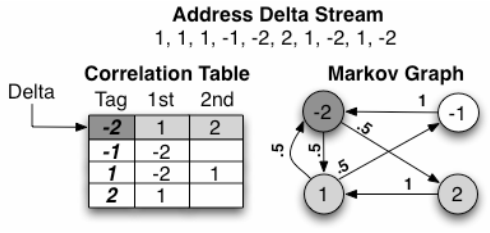

Figure 4: Distance Prefetching.

Like all correlation prefetching methods, Distance Prefetching can be modeled with a Markov graph (as shown in Figure 4). However, unlike Markov Prefetching, Distance Prefetching's state predictions are not prefetch addresses. To calculate prefetch addresses, the predicted deltas are added to the current miss address.

3. Global History Buffer Prefetching

In general, prefetch tables store prefetch history inefficiently. First, table data can become stale, and consequently reduce *prefetch accuracy* (the percent of prefetches that are actually used by the program before being evicted). Second, tables suffer from conflicts that occur when multiple access keys map to the same table entry. The most common solution for reducing table conflicts is to increase the number of table entries. However, this approach increases the table's memory requirements, and increases the percentage of stale data held in the table. Third, tables have a fixed (and usually a small) amount of history per entry. Adding more prefetch history per entry creates new opportunities for effective prefetching, but the additional prefetch history also increases the table's memory requirements and its percentage of stale data, which together can negate the advantages.

To provide more efficient prefetchers we propose an alternative prefetching structure that decouples table key matching from the storage of prefetch-related history information. The overall prefetching structure has two levels (Figure 1b).

- An *Index Table* (IT) that is accessed with a key as in conventional prefetch tables. The key may be a load instruction's PC, a cache miss address, or some combination. The entries in the Index Table contain pointers into the Global History Buffer.
- The *Global History Buffer* (GHB) is an *n*-entry FIFO table (implemented as a circular buffer) that holds the *n* most recent L2 miss addresses. Each GHB entry stores a global miss address and a link pointer. The link pointers are used to chain the GHB entries into address lists. Each address list is the time-ordered sequence of addresses that have the same Index Table key.

Depending on the key that is used for indexing the Index Table, any of a number of history-based prefetch methods can be implemented. In the following subsections we illustrate how the GHB can be used to implement correlation and stride prefetching. In addition, we illustrate more general forms of each (a total of eight prefetching methods).

To simplify the discussion and illustrate the relationship between methods, the names given to prefetching methods follow a consistent taxonomy. Each method is denoted as a pair: X/Y, where X is the key used for localizing the miss address stream and Y is the mechanism used for detecting addressing patterns. We consider two localizing methods: Program Counter (PC) and Global (G), and three detection mechanisms: Constant Stride (CS), Delta Correlation (DC), and Address Correlation (AC). Note that existing table-based methods fit into this taxonomy (i.e. Stride Prefetching is PC/CS, Distance Prefetching is G/DC, and Markov Prefetching G/AC).

3.1. Example: Markov Prefetching

We first use Markov Prefetching (G/AC) to illustrate a GHB prefetcher. See Figure 5. When an L2 cache miss occurs, the miss address indexes the Index Table. If there is a hit in the Index Table, the Index Table entry will point to the most recent occurrence of

the same miss address in the GHB. This GHB entry is also at the head of the linked list of other entries with the same miss address. For each entry in this linked list, the next entry in the FIFO ordered GHB is the miss address that immediately followed the current miss address when it occurred in the past. These "next" global miss addresses are prefetch candidates. With the bottom-to-top orientation of the GHB in Figure 5, the "next" GHB entry is immediately below the current entry. To better illustrate the method, the current memory address is shaded with a darker gray and the prefetch memory addresses are shaded with a lighter gray. In the example, prefetch candidates generated by walking the address list are *C* and *B*, the same addresses held in the conventional Markov correlation table in Figure 3.

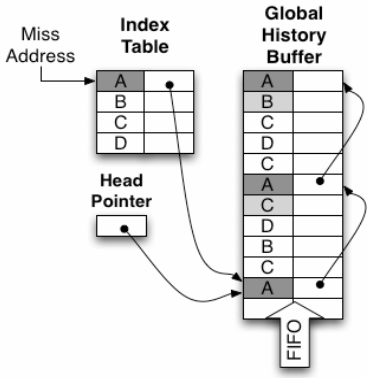

Figure 5: GHB Global / Address Correlation

3.2. Example: Stride Prefetching

To implement Stride Prefetching (PC/CS), the GHB structure detects load instructions with a constant stride. Using the PC of a load instruction as the index into the Index Table, the address list created is the sequence of addresses for the given PC. The load's strides can be calculated by computing the differences between consecutive entries in the address list. If a constant stride is detected, e.g. if the first *x* computed strides are the same, then addresses *a+s, a+2s, . . . a+ds* are prefetched -- where *a* is the current miss address, *s* is the detected stride and *d* is the prefetch degree. For this paper we use an *x* value of 2, which is consistent with conventional stride prefetchers.

3.3. Implementation

A simple state machine maintains the GHB address lists and coordinates GHB accesses to walk the address lists. For each new miss, the GHB is updated in FIFO fashion. The miss address is placed into the GHB entry pointed to by the head pointer (See Figure 5), and its link entry is given the current value in the Index Table. The Index Table link entry is then updated with the head pointer, which points to the newly added entry. Finally, the head pointer is incremented to point to the next GHB entry.

A GHB prefetch error can occur when a pointer into the GHB (either a pointer stored in the Index Table or one stored in the GHB) no longer points to the correct address sequence in the GHB. This is possible because the GHB is a circular buffer, and when an entry is overwritten, any pointers to the entry become invalid. One way to detect invalid GHB pointers is to increase the width of the GHB pointers, but only use the low-order bits of the pointers to actually index the GHB. Then, if the difference between the head pointer and another pointer is greater than the size of the GHB, the pointer is invalid. This method is not perfect, however; it is still possible for the head pointer to wrap around and cause an incorrect match. We have found that increasing the width of the pointers by four-bits (the number of bits used in our simulations) makes the probability of incorrect matches very low.

As a circular buffer, the GHB prefetching structure eliminates many of the problems associated with conventional tables (as described above). First, the GHB FIFO naturally gives table space priority to the most recent global history, thus eliminating the stale data problem. Furthermore, the GHB naturally allocates more chip-area to events that have occurred more recently and more often.

Second, the Index Table and GHB are sized separately. The Index Table only needs to be large enough to hold the working set of prefetch keys. Moreover, Index Table entries are relatively small; they contain a tag (for matching) and a single pointer into the GHB (on the order of 1-2 bytes). The GHB has greater memory requirements, but is sized independently (in the absence of table conflicts) to hold a representative portion of the miss address stream.

Last, and perhaps most importantly, as we shall see in the next two subsections, the ordered global history can be used to create more sophisticated prefetching methods.

A possible drawback to using the GHB is that collecting prefetch information requires multiple table accesses (to follow the linked lists), however, this delay is relatively small in comparison with the L2 miss delay. The table access delay is accounted for in our simulator.

3.4. Generalized Correlation Prefetching

The Markov model can serve as a basis for a number of global correlation prefetching methods. As noted earlier, the nodes are addresses and an arc connecting two nodes indicates the probability that one

address is followed by the other in the global miss address stream. Prefetch algorithms can use this information in a number of ways to predict future addresses.

In terms of the Markov graph, existing Markov (G/AC) and Distance Prefetching (G/DC) methods essentially start at a node and prefetch using one or more adjacent nodes – but only the immediately adjacent nodes. We refer to this as *width* prefetching, and it is "wired in" to the table structures that implement Markov and Distance Prefetching. In terms of the original Markov graph, however, one can also consider *depth* prefetching; i.e. beginning with the current miss address, the sequence of the most likely arcs is followed, with prefetching initiated at each node along the path. Of course, one can also consider hybrid methods that use a combination of width and depth, i.e. the sequence of the most likely arcs are prefetched, then the second most likely, and so on.

For most workloads, a problem with relying only on width is that the effective look-ahead distance is relatively short and prefetches have poor *timeliness* (whether prefetches are issued early enough to prevent processor stalls). On the other hand, depth prefetching allows the prefetcher to run farther ahead of the actual address stream [14].

The GHB method can be used for either width or depth prefetching as well as hybrid combinations. Following the address linked list alone gives width prefetching; using the sequential GHB entries beginning at a member in the address list adds depth.

Figure 6 illustrates how the GHB can prefetch depth in the *global* miss address stream using delta correlations, *Global / Delta Correlation depth* (G/DC depth) prefetching. The lighter "Deltas" box shown in the figure does not exist in GHB hardware, but is extracted by finding the difference between miss addresses in the GHB. As shown in the figure, prefetch addresses are generated by taking the miss address and accumulatively adding deltas; a valid prefetch address is created from each addition.

With the GHB approach, one can often get a better estimate of the actual Markov graph transition probabilities than with conventional correlation methods. In fact, the GHB allows a weighting of transition probabilities based on how recently they have occurred. For example, if the last five Markov transitions from state *A* were to *C, B, B, B,* and *B*, where *C* is the oldest transition, a GHB prefetching method (with a width of two) would examine the first two transitions and prefetch *B*. In contrast, a correlation table prefetching method would prefetch *B* and *C*, no matter how long ago the transition to *C* occurred. In practice, the GHB's width prefetching method results in improved prefetch accuracy (as will be shown in Section 4.4.1).

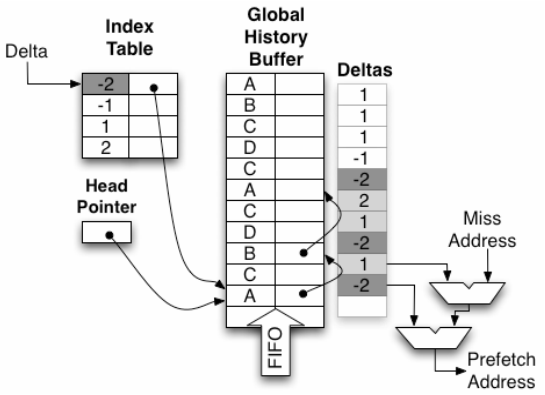

Figure 6: GHB Global / Delta Correlation

3.5. Local Delta Correlation

Localizing address streams with the PC, like Stride Prefetching does, is a very effective method for dividing the global miss address stream into separate access patterns. However, Stride Prefetching (PC/CS) is limited by the history held in its prefetching table, the previous miss address and previous stride. As a result, its most sophisticated mechanism for detecting patterns is to simply compare the current stride with the previous stride. This approach is good for most loads, but there are many loads with predictable access patterns that are not constant strides, particularly loads that use pointer arithmetic to access a data structure. For example, consider the address and delta stream below.

Addresses	0	1	2	64	65	66	128	129
Deltas		*1*	*1*	*62*	*1*	*1*	*62*	*1*

This access pattern is representative of a load that accesses the first three words of each column in a two-dimensional array. Such a pattern can trick constant stride prefetching mechanisms into generating superfluous prefetches. In this example, the short bursts of unit strides will cause a constant stride method to prefetch down an incorrect unit stride address stream.

In contrast, the GHB contains the actual sequence of a load's miss addresses (up to the limit of the GHB size). This information can be used for detecting delta access patterns within a load's address stream, and prefetching down the non-stride, but regular, delta access pattern, like the one shown above.

This new GHB method, *Program Counter / Delta Correlation* (PC/DC), uses delta pairs (two consecutive deltas) as the correlation key. With delta-pairs, the method can accurately describe the entire access pattern above with three correlations. See Table 1. In the example, the two most recent deltas are 62 and 1. If the address sequence is searched in reverse order for the same delta pattern, it is first found at (2, 64, 65).

When the delta pair (62, 1) appeared previously, the next deltas were 1, 62, 1, and 1, therefore, if the prefetch degree is four, addresses 130, 192, 193, and 184 would be prefetched.

Table 1: Example Address Stream Correlations

Local Correlation Key (most recent delta pair)	Prefetch Prediction (4 subsequent deltas)			
(1, 1)	*62*	*1*	*1*	*62*
(1, 62)	*1*	*1*	*62*	*1*
(62, 1)	*1*	*62*	*1*	*1*

4. Performance Evaluation

4.1. Simulation Methodology

To evaluate GHB prefetch methods we use a subset of the SPEC CPU2000 benchmark suite. Tables 2 and 3 contain the benchmarks and their IPC improvements with an ideal L2 cache, reflecting the maximum prefetch potential. The subset includes all the SPEC benchmarks with a maximum prefetch potential of at least 5%, except for the floating point benchmark *sixtrack*. We had to exclude *sixtrack* because it intermittently generated unsupported system calls, which were very difficult to consistently reproduce and debug. For simulations, the first billion instructions are skipped, and data is collected for the next billion instructions.

Table 2: SPEC FP Subset with Ideal L2 IPC Improvement.

Benchmark	IPC (%)
ammp	815%
art	238%
wupwise	204%
swim	136%
lucas	135%
mgrid	56%
applu	36%
galgel	30%
apsi	20%

Table 3: SPEC INT Subset with Ideal L2 IPC Improvement.

Benchmark	IPC (%)
Mcf	319%
twolf	104%
vpr	78%
parser	40%
gap	22%
bzip2	19%

SimpleScalar 3.0 [1] was used for collecting performance data. The simulator configuration is detailed in Table 4.

To eliminate the need for additional prefetch structures, prefetched lines are placed directly into the L2 cache. Before a prefetch is issued to the memory subsystem the L2's tag array is probed to ensure the prefetch address is not already in the cache. To keep prefetched (but not yet accessed) lines from modifying the "natural" L2 miss address stream, one bit prefetch tags are added to the L2 cache lines. When a prefetched line is written into the L2 cache, its prefetch tag is set; when a cache access hits a prefetched line with a set prefetch tag, the prefetch tag is cleared, and the access's memory address is sent to update the prefetch structures as if it were an L2 cache miss.

For timing simulations, each access to the Index Table and GHB memory arrays is assumed to have a one cycle read latency, which is reasonable for relatively small tables. If an L2 miss occurs while a previous prefetch query is being serviced by the GHB state machine, the previous query is aborted and the new L2 miss is handled.

Table 4: Simulator Configuration

Issue Width	4 instructions
Load Store Queue	64 entries
RUU Size	128 entries
Level 1 D-Cache	16KB, 2-way set associative
Level 1 I-Cache	16KB, 2-way set associative
Level 2 Cache	512KB, 2-way set associative
Memory Latency	140 Cycles

For performance evaluation, we focus on the Global / Delta Correlation methods (Distance Prefetching and GHB G/DC) and PC local methods (conventional Stride Prefetching, GHB PC/CS, and GHB PC/DC). We do not give Markov Prefetching (G/AC) results. Although Markov Prefetching is one of the basic approaches, G/AC performance is generally worse than G/DC and requires much more storage (on the order of megabytes).

4.2. Table Configurations

An initial set of simulations (not shown) was used for determining table sizes for each prefetching method. For these simulations, the prefetch degree for each method was held constant at four. For conventional Distance Prefetching (G/DC), this meant the number of correlation pairs held in each table entry was also fixed at four. Then the number of table entries was varied to find the optimal (or near optimal) table size. For the GHB methods, the numbers of Index Table and GHB entries were varied independently. An optimal number of GHB entries was first found by using an overly large Index Table size (128K entries) and varying the GHB size. Then using this optimal GHB size, an optimal number of Index Table entries was determined. To compare the performance of different sized tables we use the IPC Improvement (with respect to no prefetching) harmonic mean of the benchmarks.

Although we study performance for a number of GHB methods we found that one table configuration was optimal (or nearly so) for all the GHB G/DC methods and one table configuration was optimal for

all GHB PC local methods (i.e. PC/CS and PC/DC). Table 6 summarizes the table configurations chosen for each method. When calculating table size the 32-bit tags in the conventional tables and in the Index Table are included. The size is rounded up to the nearest kilobyte.

Table 6: Table Configurations

Prefetching Method	Table Configuration	Size
Conventional Distance Prefetching (G/DC)	512 table entries	18KB
GHB G/DC	512 IT entries x 512 GHB entries	8KB
Conventional Stride Prefetching (PC/CS)	256 table entries	6KB
GHB PC/CS and PC/DC	256 IT entries x 256 GHB entries	4KB

In general, the size for the GHB methods is smaller then their conventional counterparts with a similar configuration.

4.3. GHB Prefetch Performance

The GHB-based prefetchers are evaluated and compared with their conventional table-based counterparts. The first set of graphs (Figures 7 and 11) in Subsections 4.3.1 and 4.3.2 compare the performance (IPC Improvement) of conventional table methods with their GHB counterparts (each with a prefetch degree of four) on a per benchmark basis. A second set of graphs (Figures 8 and 12) illustrate performance (IPC Improvement harmonic mean) of a range of different prefetch degrees from 1 to 16. A third set of graphs (Figures 9 and 13) show the arithmetic mean increase in memory traffic per instruction (with respect to no prefetching).

4.3.1. Global / Delta Correlation

This subsection compares conventional Distance Prefetching (*Table G/DC (width*)), with three GHB G/DC implementations described in subsection 3.4: depth prefetching (*GHB G/DC (depth)*), width prefetching (*GHB G/DC (width)*) and a combination of width and depth prefetching (*GHB G/DC (hybrid)*). See Figures 7, 8, and 9.

The hybrid prefetching method in effect has two prefetch degree components, a width and depth component. For this study we make them equal and state the single width/depth number as the overall degree of the method. Note that this terminology abuses our definition of prefetch degree, however, e.g. a hybrid method with a prefetch degree of four can actually generate sixteen prefetch requests at a time.

For the hybrid method with a large prefetch degree, the width component or the product of the width and depth components are only weakly related to the amount of data actually prefetched. Consider the 16 depth x 16 width hybrid method, in theory, this method performs 256 prefetches, but very often the actual number of prefetches will be much less for a number of reasons. First, before 256 memory addresses are prefetched the method will likely be preempted by another L2 miss. Second, because the method is prefetching 16 nodes (depth) of the 16 most recent Markov chains (width), most prefetches in subsequent Markov chains will overlap. Third, the method will likely run out of GHB history before all the prefetches are done. For these reasons, the stated prefetch degree is more closely related to prefetch depth than prefetch width.

For a prefetch degree of four (Figure 7), GHB G/DC hybrid prefetching outperforms conventional Distance Prefetching on all benchmarks except *twolf* and *vpr*. Overall, GHB G/DC hybrid prefetching outperforms conventional Distance Prefetching by 13%. The majority of GHB G/DC hybrid's overall IPC Improvement is a result of a few benchmarks (i.e. *ammp*, *swim, lucas, mgrid*, and *mcf*), where the hybrid's additional look-ahead and *coverage* (the percent of memory references prefetched rather than demand fetched) pay off. In the case of *twolf* and *vpr*, all prefetching methods degrade performance, and GHB G/DC hybrid degrades performance the most.

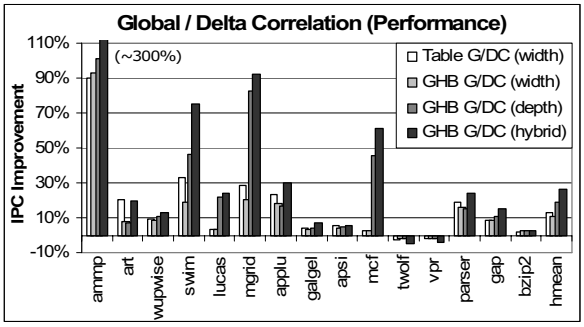

Figure 7: Global / Delta Correlation (prefetch degree four) IPC Improvement per benchmark.

Turning to the results for prefetch degrees 1 through 16 (Figures 8 and 9), conventional Distance Prefetching outperforms the GHB G/DC width by less than 4%, but has 90% more memory traffic. These results tend to support our claim that conventional correlation tables suffer from stale data, and consequently, poor prefetch accuracy. For our simulator configuration, the additional prefetches enhance prefetch coverage enough to improve overall performance by a small amount, however, on a system with constrained mem-

ory bandwidth, the additional memory traffic would likely degrade performance.

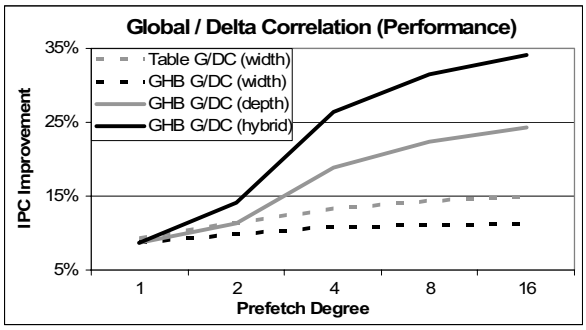

Figure 8: Global / Delta Correlation IPC Improvement for prefetch degrees 1 through 16.

To better illustrate the behavior of stale table data, we tracked the age of each entry in the Distance Prefetching correlation table. The age of an entry is the number of cycles since the entry was last touched. A logarithmic scale is used to form age groups, e.g. the first age group is less than 16 cycles, the second age group is between 16 and 256 cycles, etc. The simulator used the age of the correlations to monitor the number of prefetches generated from each age group. When a prefetch was generated, the age of the correlation was included with the prefetch request, allowing the simulator to monitor how many prefetches from each age group result in a cache hit. With this data the accuracy of prefetches from each age group was calculated.

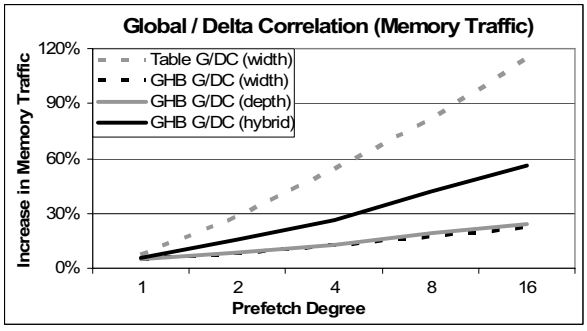

Figure 9: Global / Delta Correlation Increase in Memory Traffic for prefetch degrees 1 through 16.

Prefetches generated by entries less than 4K cycles old are 10 times more accurate than prefetches generated by entries older than 16M cycles. See Figure 10. Furthermore, most prefetches are from entries that are between 64K and 1M cycles old, and their prefetch accuracy is less than half the accuracy of prefetches generated by entries less than 4K cycles old.

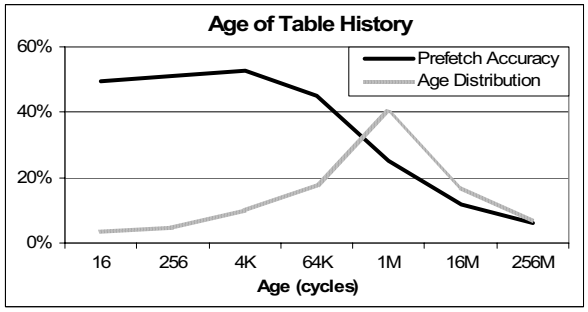

Figure 10: Age distribution of Table History that generates a prefetch and prefetch accuracy per age group.

Also illustrated by Figure 8, the G/DC width prefetching methods (both conventional Distance Prefetching and GHB G/DC width) do not have the look-ahead to capture the full potential of G/DC prefetching. Depth prefetching outperforms width prefetching by 13% (at a degree of 16), while using the same size tables and has approximately the same amount of memory traffic. Hybrid prefetching outperforms depth prefetching by an additional 10% (outperforms width prefetching by 23%).

Depth prefetching does not perform as well as hybrid prefetching for two reasons. First, correlations often occur close to the head of the GHB, and second, depth prefetching cannot achieve the same coverage as hybrid prefetching. When a correlation is close to the head of the GHB, the depth method runs out of history and terminates before prefetching its entire prefetch degree. Hybrid prefetching resolves this issue by prefetching depth until it reaches the head of the GHB, then prefetches the next Markov chain, which is farther from the head. It is possible to modify the depth method to search the GHB for a Markov chain long enough or delay updating the Index Table so there will always be enough history to prefetch, however it is just as easy to use the hybrid method and get the additional advantage of better prefetch coverage.

A drawback to hybrid prefetching is increased memory traffic. See Figure 9. The hybrid method consumes 30% more memory traffic than the depth method, however, when compared to conventional Distance Prefetching, which has 90% more memory traffic than GHB G/DC width prefetching, the hybrid prefetching method's memory traffic is relatively low.

4.3.2. PC Local Prefetching

In this subsection we compare conventional Stride Prefetching (*Table PC/CS*) with *GHB PC/CS* and *GHB PC/DC* prefetching . See Figures 11, 12, and 13.

For a prefetch degree of four, GHB PC/DC consistently performs better than constant stride prefetching, approximately 3% overall; the only exceptions are

benchmarks *galgel* and *apsi.* In the case of *galgel* and *apsi,* GHB PC/DC is outperformed by 1% and 5% respectively.

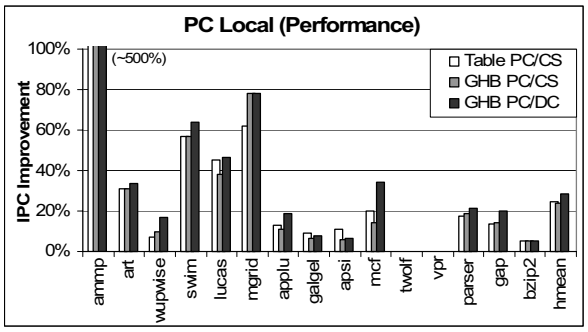

Figure 11: PC Local methods (PC/CS and PC/DC) (prefetch degree four) IPC Improvement per benchmark.

Comparing Figures 7 and 11, GHB PC/DC and G/DC (hybrid) have similar overall performance, but each excels on different benchmarks, i.e. GHB G/DC (hybrid) outperforms PC/DC by over 10% on *swim*, *mgrid*, *applu*, and *mcf*, and of the remaining benchmarks PC/DC tends to perform as well or better.

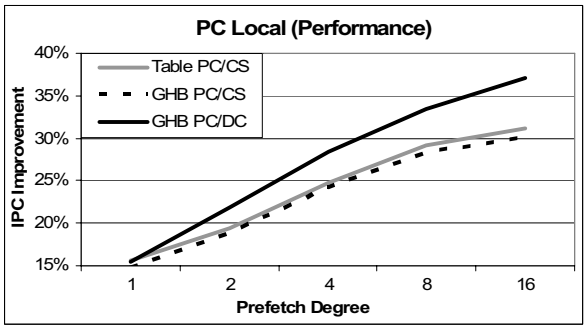

Figure 12: PC Local methods (PC/CS and PC/DC) IPC Improvement for prefetch degrees 1 through 16.

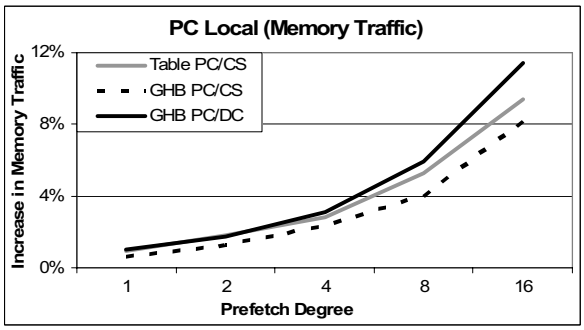

Figure 13: PC Local methods (PC/CS and PC/DC) Increase in Memory Traffic for prefetch degrees 1 through 16.

At a prefetch degree of sixteen, conventional Stride Prefetching (PC/CS) performs 1% better and consumes 1% more traffic than GHB PC/CS (Figures 12 and 13). In effect, the two provide equivalent performance. Figure 13 shows the PC local methods do not benefit from the GHB's ability to reduce stale data as much as G/DC methods (Figure 9). For conventional Stride Prefetching, table entries are associated with a specific load instruction. Even if a table entry has not been accessed in a long period of time, it is less likely that a load instruction's behavior will have changed since it was last accessed. In general, PC local methods have lower memory traffic than G/DC (Figures 9 and 13).

On the other hand, the GHB PC/DC prefetching method outperforms conventional Stride Prefetching (Table PC/CS) by 6% and has 3% more traffic (for a prefetch degree of sixteen). These results show that PC/DC prefetching can prefetch the same access patterns as constant stride prefetching, and gets additional performance from its ability to prefetch the more complex delta access patterns. Naturally, there is a percentage of prefetches for delta patterns that are incorrect, resulting in an increase in memory traffic.

5. Conclusions

In this paper we propose a new Global History Buffer (GHB) prefetching structure built and illustrate how it can improve existing prefetch algorithms. Collectively, the new GHB prefetching methods are shown to perform as well or better than their conventional counterparts on 14 of the 15 benchmarks studied (Figures 7 and 11). The first GHB method, GHB G/DC hybrid, is shown to have a 20% IPC Improvement over its conventional counterpart, Distance Prefetching, and it can reduce memory traffic by 90%. The second GHB method, GHB PC/DC, is shown to have a 6% IPC improvement over its conventional counterpart Stride Prefetching.

In general, GHB prefetchers have three basic advantages with respect to conventional table-based prefetchers. First, the global history buffer reduces stale table data, thus improving prefetch accuracy and reducing prefetch memory traffic. Second, the global history buffer contains a more complete (and intact) picture of cache miss history, creating opportunities to design more effective prefetching methods (i.e. GHB G/DC hybrid and PC/DC). Third, the new structure's size is generally smaller than conventional tables

A potential disadvantage is the need to make multiple table accesses, but for L2 (or L3) caches, the time required for these accesses is relatively low with respect to the entire miss latency. Furthermore, a single

series of linked list accesses can invoke a number of prefetch requests.

Another interesting feature of the GHB, which we have thus far not discussed, is that it provides a way of implementing a number of prefetch methods with a single underlying structure. As seen in Subsections 4.3.1 and 4.3.2, there is no single prefetching method and prefetch degree that result in an optimal performance to memory traffic ratio for all benchmarks. As a solution to this problem, the global history buffer can be used as a unified prefetching structure that can be dynamically configured to implement a number prefetching methods, depending on the program or program phase. We plan to investigate this possibility in future research.

6. Acknowledgements

This research was funded by an Intel Undergraduate Research Scholarship, a University of Wisconsin Hilldale Undergraduate Research Fellowship, and by the National Science Foundation under grants CCR-0311361 and EIA-0071924. The authors would also like to thank Anthony Wojciechowski and Nick Lindberg for their helpful suggestions.

7. References

[1] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.org.

[2] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. October 2001. http://www.cs.wisc.edu/multifacet/misc/spec2000cachedata/

[3] M. J. Charney and T. R. Puzak, Prefetching and memory system behavior of the SPEC95 benchmark suite, *IBM Journal of Research and Development*, 31(3), 1997.

[4] T. Chen and J. Baer. Effective hardware based data prefetching for high-performance processors. *IEEE Transactions on Computer Systems*, 44(5):609–623, May 1995.

[5] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[6] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. IEEE *Transactions on Computer Systems*, 48(2):121–133, 1999.

[7] Z. Hu, M. Martonosi, S. Kaxiras, TCP Tag Correlating Prefetchers, In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, 2003.

[8] N. Jouppi. Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, 1990.

[9] G. B. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceeding of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[10] S. Kim, A. Veidenbaum, Stride-directed Prefetching for Secondary Caches, In *1997 International Conference on Parallel Processing*, 1997.

[11] A.C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. *In Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[12] Subbarao Palacharla and R.E. Kessler. Evaulating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, May 1994.

[13] A.J. Smith, Sequential Program Prefetching in Memory Hierarchies*, IEEE Transactions on Computers*., Vol. 11, No. 12, pp.7-21, Dec. 1978.

[14] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching, In *Proceedings of the 29th Annual International Symposium on Computer Architectur*e, May 2002.

[15] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, POWER4 System Microarchitecture, *IBM Technical White Paper*, 2001.

[16] S. VanderWiel and D. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, June 1999.

[17] Z. Wang, D. Burger, K. McKinley, S. Reinhardt, C. Weems, Guided Region Prefetching: A Cooperative Hardware/Software Approach, In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.