Data Cache Prefetching Using a Global History Buffer

By organizing data cache prefetch information in a new way, a GHB supports existing prefetch algorithms more effectively than conventional prefetch tables. It reduces stale table data, improving accuracy and reducing memory traffic. It contains a more complete picture of cache miss history and is smaller than conventional tables.

Kyle J. Nesbit
James E. Smith
University of Wisconsin-Madison

Over the past couple of decades, trends in both microarchitecture and underlying semiconductor technology have significantly reduced microprocessor clock periods. Meanwhile, the technology trend in main memories has been a move toward higher densities rather than significantly reduced access times. Together, these trends have significantly increased relative main-memory latencies as measured in processor clock cycles. To avoid large performance losses caused by long memory access delays, microprocessors rely heavily on a hierarchy of cache memories. But cache memories are not always effective, either because they are not large enough to hold a program's working set, or because memory access patterns don't exhibit behavior that matches a cache memory's demand-driven, line-structured organization.

To partially overcome cache memories' limitations, the cache memory controller can predict memory addresses likely to be accessed soon and then prefetch the data into the cache. Basic methods prefetch sequential lines; that is, they access cache lines at addresses immediately following the line currently being accessed. Simple sequential methods prefetch after every cache miss, or they prefetch cache lines immediately following earlier prefetched lines that the processor has actually used.[1]

To be effective, prefetching must be timely; a prefetch request must occur far enough in advance that the prefetch data is available at the time it is needed. On the other hand, overly aggressive prefetching can actually reduce performance by wasting limited memory bandwidth and other hardware resources. Consequently, more advanced prefetch algorithms prefetch only after detecting a pattern of miss addresses, providing some assurance that future prefetches will be useful.[2] Then, if prefetches are indeed successful, the algorithm can increase the prefetch degree (the maximum number of cache lines prefetched in response to a single prefetch request) until the latency of a main-memory access is completely hidden.

The more advanced prefetch methods use tables to record history information about memory-addressing behavior.[3-6] Figure 1a illustrates conventional table-based methods. A prefetch table can contain stride information, a *stride* being a constant value that separates members of an address sequence. Or the table can contain information describing more-complex access patterns, as in Markov prefetching.[4] The prefetcher accesses the table

Published by the IEEE Computer Society

with a key, such as a load instruction's program counter or a miss address. Then, the prefetcher uses history information read from the table to predict future miss addresses and launch prefetch requests.

Although they are simple, conventional table-based methods are relatively inefficient because they reserve a fixed amount of table space per prefetch key. Depending on the keys that occur, the history information in some table entries might go unused for a very long time and become stale—that is, the information no longer reflects current conditions. Accessing a stale table entry can trigger ineffective prefetches.

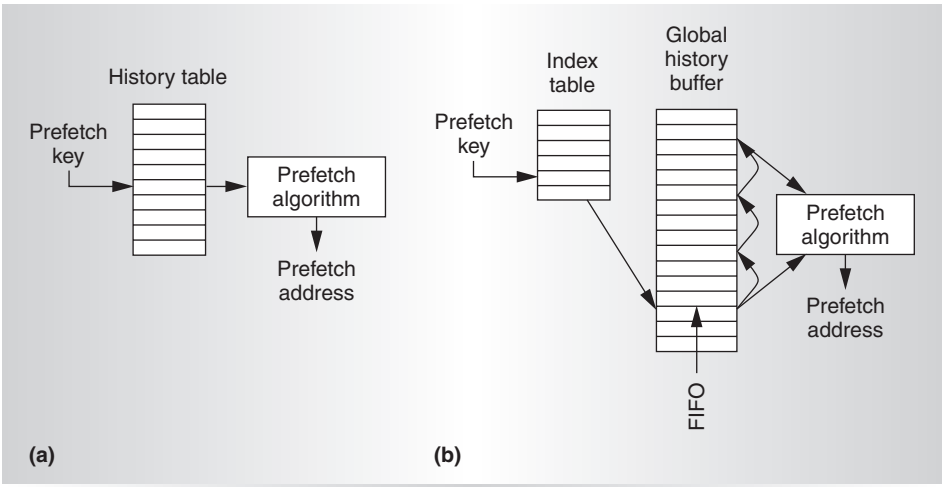

Figure 1. Table-based data cache prefetch methods: conventional prefetch table (a); global history buffer prefetch structure (b).

We propose an alternative structure, shown in Figure 1b, for holding prefetch history. In this structure, a fixed-length FIFO table, the global history buffer (GHB) holds cache miss addresses. All miss addresses enter the table in the order they occur; in the figure, they enter at the bottom and exit from the top. GHB information associated with a given prefetch key is organized as a linked list, which is accessed indirectly via a hash table. This method automatically reduces the amount of stale address history and allows a more accurate reconstruction of miss address patterns in the context in which they occur. As a result, a designer can implement enhanced prefetching algorithms based on strides as well as more-complex access patterns.

Like most recent research on data cache prefetching, our work focuses on the data cache level closest to main memory because modern out-of-order superscalar processors more easily tolerate misses to the other cache levels. In our research, we focus on systems with two cache levels, L1 and L2, and prefetch into L2. Of course, we can extend the GHB method to an L3 cache, if present.

Table-based prefetching

Before proceeding with GHB-based prefetch algorithms, we first review conventional table-based versions, generally illustrated in Figure 1a.

Stride prefetching

Conventional stride prefetching uses a table to store stride-related history information for individual load instructions.[3] A load instruction's program counter (PC) is the prefetch key that indexes the table. Each table entry holds the load's most recent previous stride (the difference between its two most recent preceding miss addresses), the most recent previous miss address (to allow computation of the next stride), and state information describing the consistency of the load's recent stride behavior. Following a cache miss, if the prefetch algorithm detects a constant stride pattern, it triggers prefetches for addresses $a + s$, $a + 2s$, …, $a + ds$, where a is the load's current address, s is the detected stride, and d is the prefetch degree. More aggressive prefetch implementations use a higher value for d.

Correlation prefetching

Correlation prefetching can prefetch more-complex patterns than strides. Markov prefetching is a correlation method that uses a history table to record consecutive miss addresses.[4] The prefetch key is a memory address (irrespective of the load's PC) that misses in the cache. Each table entry holds a list of miss addresses that have immediately followed the prefetch key's address in the past. When a cache miss occurs, the miss address indexes the correlation table, and the member(s) of the table entry's address list are prefetched, with the most recent miss address first.

Distance prefetching is a generalization of Markov prefetching.[5] Originally proposed for prefetching TLB entries, distance prefetching can easily be adapted to prefetching cache lines. This adaptation uses an *address delta*, the distance between two consecutive miss addresses, as the prefetch key into the correlation table. Each correlation table entry holds a list of deltas that have followed the prefetch key's delta in the past. The algorithm then uses the delta list to compute prefetch addresses by adding the deltas to the current miss address. Distance prefetching is a more compact form of Markov prefetching because one delta correlation can represent many miss address correlations. Also, because it uses deltas rather than absolute addresses, distance prefetching can trigger successful prefetches for miss addresses that have not occurred in the past.

Conventional prefetch tables store prefetch history inefficiently. First, table data can become stale and consequently reduce prefetch accuracy (the percentage of prefetches for data the program actually uses). Second, tables suffer from conflicts when multiple prefetch keys hash to the same table entry. A common solution for reducing table conflicts is to increase the number of entries, but this approach increases the table's memory requirements and the percentage of stale data held in the table. Third, tables hold a fixed, usually 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 the amount of stale data.

GHB prefetching

For more efficient prefetchers, our alternative structure decouples prefetch key matching from the storage of prefetch-related history information. As Figure 1b shows, the overall structure has two levels:

- *Index table.* Prefetch algorithms access the index table with a key as in conventional prefetch methods. The key can be a load instruction's PC, a cache miss address, or a hashed combination of the two. Index table entries contain pointers into the GHB.
- *Global history buffer.* The 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 chain the GHB entries into address lists. Each address list is a time-ordered sequence of addresses with the same index table prefetch key.

Depending on the key used for hashing into the index table, a designer can implement any of several history-based prefetch methods. Here, we describe the GHB's use for correlation and stride prefetching and more general forms of each.

As a circular buffer, the GHB prefetching structure eliminates many problems associated with conventional tables. First, the GHB FIFO naturally gives table space priority to the most recent history, thus eliminating the stale-data problem. Second, the index table and the GHB are sized separately. The index table must only be large enough to hold the working set of prefetch keys. Moreover, index table entries are relatively small, containing a tag (for hash index matching) and a single pointer into the GHB (about 1 or 2 bytes). The GHB is larger, with a size chosen to hold a representative portion of the miss address stream. Last, and perhaps most important, a designer can use the ordered global history to create more-sophisticated prefetching methods than conventional stride and correlation prefetching.

A possible drawback of 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. Our performance evaluation takes the table access delay into account.

To simplify the discussion and illustrate relationships between prefetch methods, we follow a consistent taxonomy in naming the methods. We denote each method as a pair X/Y, in which X is the prefetch key and Y is the mechanism for detecting addressing patterns. We consider two prefetch keys: program counter (PC) and global addresses (G)—miss addresses that are independent of PC. We consider three detection mechanisms: constant stride (CS), address correlation (AC), and delta correlation (DC). Conventional table-based methods fit into this taxonomy—stride

prefetching is PC/CS, Markov prefetching is G/AC, and distance prefetching is G/DC.

Markov prefetching

We first use Markov prefetching (G/AC) to illustrate a GHB implementation. Figure 2 shows the structure of a GHB G/AC prefetcher. 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 FIFO-ordered entry in the GHB is the miss address that immediately followed the current miss address in the past. These "next" miss addresses, one for each linked list member, are prefetch candidates. With the GHB's bottom-to-top orientation in Figure 2, the "next" GHB entries (shaded light gray) are immediately below the current miss's linked list entries (shaded dark grey). Therefore, in this example, the prefetch candidates generated by walking the address linked list are C and B.

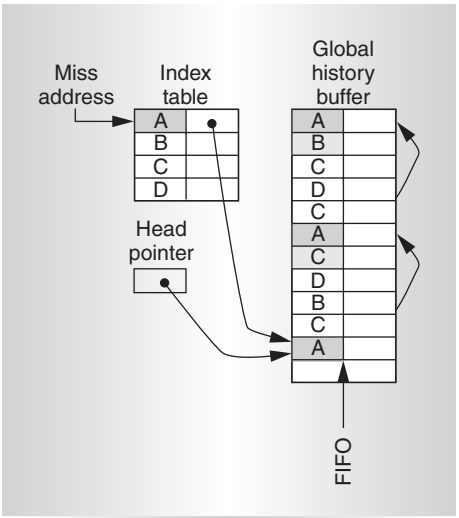

Figure 2. GHB global/address correlation (G/AC) prefetcher.

Stride prefetching

In stride prefetching (PC/CS), a load instruction's PC hashes into the index table, and the GHB address list is the sequence of recent miss addresses for the given PC. The prefetcher calculates the load's strides by computing the differences between consecutive entries in the linked list. If the prefetcher detects a constant stride—for example, if the first x computed strides are the same—the prefetcher generates prefetch requests for the constant-strided address stream. In this article, we use an x value of 2, which is consistent with most conventional stride prefetchers.

Generalized correlation prefetching

Existing Markov (G/AC) and distance prefetching (G/DC) methods prefetch only according to immediate successor correlations. Markov methods prefetch using w addresses that immediately followed the current address in the past, and distance methods prefetch according to w address deltas that have immediately followed the current delta. We refer to this method as *width prefetching*. The value w is "wired" into the table structures that implement Markov and distance prefetching—that is, it depends on the number of elements in each table entry. For most workloads, a problem with relying only on width is that the effective look-ahead is relatively short and prefetches have poor timeliness.

A variation is depth prefetching. In this method, the prefetching mechanism begins with the current miss address and accesses the GHB sequentially, triggering a series of prefetches of length d. In Figure 2, using a d value of 2 would result in the prefetching of both C and D by the linked list's middle element. Depth prefetching lets the prefetcher run farther ahead of the actual address stream.[6] There are also hybrid methods that use a combination of width and depth. In Figure 2, if w is 2 and d is 2, blocks C and D will be prefetched, followed by B and C (although the second prefetch to C is redundant and will be filtered out).

In practice, the GHB correlation prefetching method improves prefetch accuracy by ignoring old prefetch data. Consider an example (not illustrated by a figure): With Markov prefetching, if the successors of A were B, B, … B, and C, where C is the oldest successor, a GHB prefetching method with a w of 2 would examine the first two successors and prefetch only B. In contrast, a conventional table prefetching method would prefetch B and C, no matter how long ago successor C occurred.

Table 1. Example address and delta stream.

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

Table 2. Address stream correlations for example in Table 1.

Most recent delta pairs	Prefetch prediction (4 subsequent deltas)			
(1, 1)	62	1	1	62
(1, 62)	1	1	62	1
(62, 1)	1	62	1	1

Table 3. SPEC benchmarks with an ideal L2 instructions-per-cycle (IPC) improvement greater than 5 percent.

SPECfp	SPECint
ammp	mcf
art	twolf
wupwise	vpr
swim	parser
lucas	gap
mgrid	bzip2
applu	
galgel	
apsi	

PC delta correlation

Using load instruction PC values as prefetch indexes is a very effective method of dividing the 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 the previous stride. As a result, the most sophisticated way to detect patterns is simply to compare the current stride with the previous stride. This approach is good for most loads, but some loads have predictable access patterns that are not constant strides. For example, consider the address and delta stream shown in Table 1. This example access pattern is representative of a load that accesses the first three fields in an array of C-style *structs*. 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 GHB size limit). The prefetcher can use this information to detect delta access patterns within a load's address stream and to prefetch down a nonstride, but regular, delta access pattern like the example.

Our new GHB method, program counter/delta correlation (PC/DC), correlates on delta pairs (two consecutive deltas). This method can accurately describe the entire access pattern of our example with three delta pairs. Table 2 lists the correlations for the example address stream. The two most recent deltas in the example are 62 and 1. Following the miss to address 129, if the prefetcher searches the address sequence in the GHB in reverse order for the same delta pattern (62 and 1), it finds it first 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 4, addresses 130, 192, 193, and 194 will be prefetched.

Performance evaluation

To evaluate GHB prefetch methods, we used a subset of the SPEC CPU2000 benchmark suite. The subset includes all the SPEC benchmarks that have a performance improvement of at least 5 percent when simulated with an ideal L2 cache that always hits. These benchmarks have a potential for improvement via prefetching. Table 3 shows the subset.

Simulation methodology

For simulations, we skipped the first billion instructions and collected data for the next billion instructions. We used a modified version of SimpleScalar 3.0 with a more detailed memory system for collecting performance data.[7] Table 4 details the simulator configuration. To eliminate the need for additional prefetch structures, the algorithm places prefetched lines directly into the L2 cache.

For timing simulations, we assumed that each access to the index table and GHB memory arrays has a one-cycle read latency, which is reasonable for relatively small tables. If an L2 miss occurs while the GHB state machine is servicing a previous prefetch query, the prefetcher aborts the previous query and handles the new L2 miss.

Table 4. Simulator configuration.

Issue width	4 instructions
Load/store queue	64 entries
RUUsize	128 entries
Level 1 D-cache	16-Kbyte, 2-way set-associative
Level 1 I-cache	16-Kbyte, 2-way set-associative
Level 2 cache	512-Kbyte, 4-way set-associative
Memory latency	140 cycles

Table 5. Table configurations. (IT: index table)

Prefetching method	Table configuration	Size (Kbytes)
Conventional distance prefetching (G/DC)	512 table entries	18
GHB G/DC	512 IT entries × 512 GHB entries	8
Conventional stride prefetching (PC/CS)	256 table entries	6
GHB PC/DC	256 IT entries × 256 GHB entries	4

For performance evaluation, we focus on G/DC methods (both conventional distance prefetching and GHB implementations) and PC-indexed methods (conventional stride prefetching (PC/CS) and GHB PC/DC). We do not give Markov prefetching (G/AC) results because G/AC performance is generally worse than G/DC and requires far more table storage (on the order of megabytes).

As explained earlier, the GHB G/DC hybrid prefetching method has two prefetch degree components: width and depth. For this study, we made them equal and state a single *w*, *d* number as the method's overall degree. This terminology is at odds with the strict definition of prefetch degree; for example, a hybrid method with a prefetch degree of 4 can actually generate up to 16 prefetch requests at a time. For hybrid methods with a large prefetch degree, however, the product of the width and depth components is only weakly related to the amount of data actually prefetched. Such methods typically generate fewer prefetches because of redundant prefetch addresses (caused by overlaps in the GHB) and a high likelihood of preemption by another prefetch request. Consequently, because hybrid prefetching proceeds depth first, the stated prefetch degree is more closely related to prefetch depth than prefetch width.

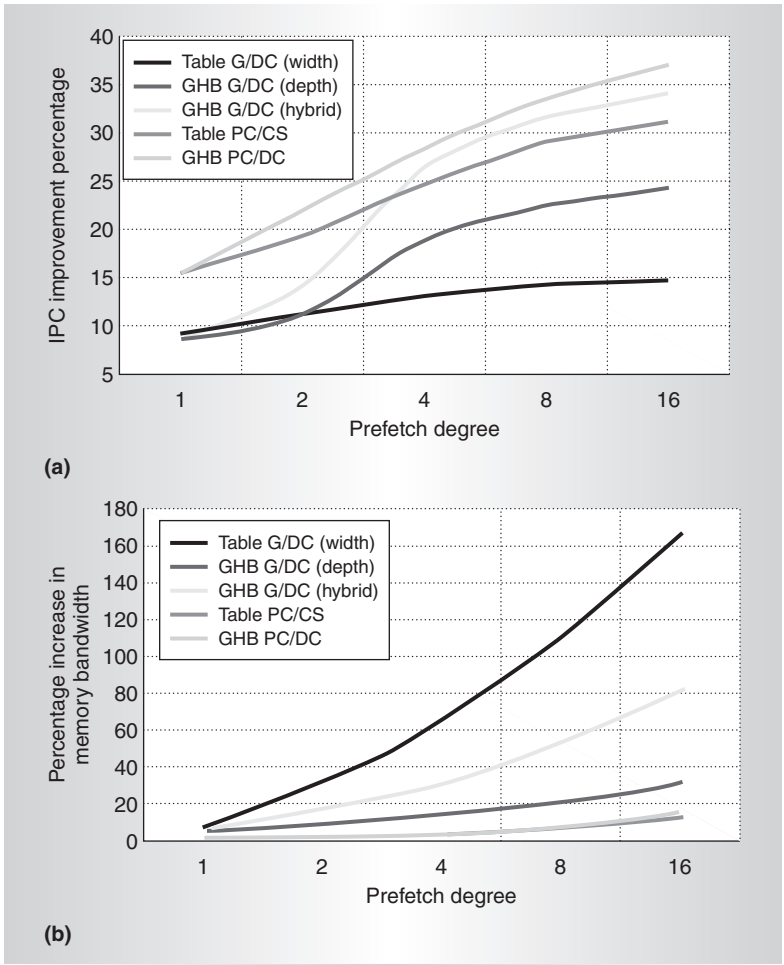

Figure 3. Performance comparisons: harmonic mean of IPC improvement for the Spec benchmark subset in Table 2 (a), and arithmetic mean of increase in memory traffic for the same Spec benchmark subset (b).

Table configurations

We used an initial set of performance simulations to determine optimal table sizes for each prefetching method. For these simulations, we held the prefetch degree constant at 4 and varied the table configurations over a wide range. Table 5 summarizes the table configurations we chose for each method. When calculating table size, we included the 32-bit tags in the conventional tables and in the index table. In general, sizes for the GHB methods are smaller than those of their conventional counterparts with similar configurations.

GHB prefetch performance

Figure 3 compares the GHB-based prefetching methods with their conventional table-based counterparts (labeled "table" in the figure). Figure 3a shows average performance improvement (measured as the harmonic mean of IPC) over a range of prefetch degrees from 1 to 16, and Figure 3b shows the arithmetic mean increase in memory traffic per

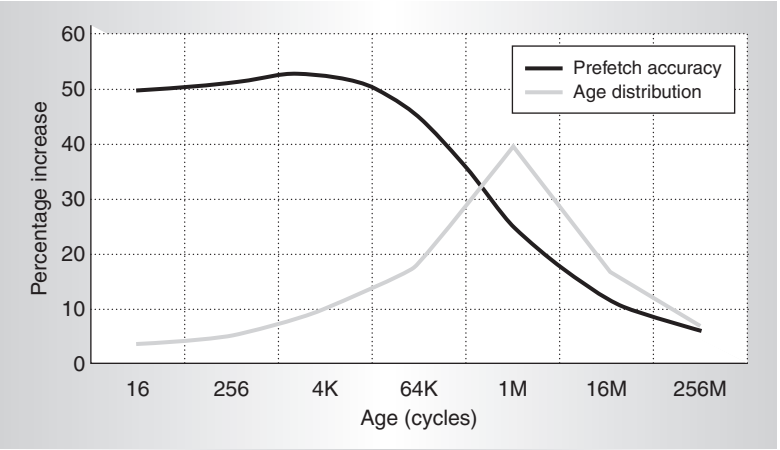

Figure 4. Age distribution of table history that generates a prefetch and prefetch accuracy per age group. The left axis represents the percentage of accurate prefetches (accuracy) and the percentage of prefetches per age group (age distribution).

instruction (with respect to no prefetching).

Considering G/DC methods first, it is apparent that conventional distance (G/DC width) prefetching does not have sufficient look-ahead to capture G/DC's full potential. Depth prefetching outperforms width prefetching by 10 percent (at a degree of 16). Hybrid prefetching outperforms depth prefetching by an additional 10 percent (and outperforms width prefetching by 20 percent).

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 to its entire prefetch degree. Hybrid prefetching resolves this problem by prefetching depth until it reaches the head of the GHB, and then prefetching the next depth chain, which is farther from the head.

A drawback of hybrid prefetching is increased memory traffic. The hybrid method consumes 50 percent more memory traffic than the depth method, but picking more appropriate depth and width components for the given workload (that is, choosing depth and width components that are not equal) can reduce hybrid memory traffic. On the other hand, compared to conventional distance prefetching, which has 140 percent more memory traffic than the depth method, the hybrid method's memory traffic is relatively low. These results support our claim that conventional correlation tables suffer from stale data and, consequently, poor prefetch accuracy. On a system with tightly constrained memory bandwidth, the additional memory traffic would likely degrade performance.

To better illustrate the behavior of stale table data, we tracked each entry's age in the distance-prefetching correlation table. An entry's age is the number of cycles since the entry was last touched. The tracking method employs a logarithmic scale to form age groups; the first age group is less than 16 cycles, the second age group is between 16 and 256 cycles, and so on. 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 correlation's age was included with the prefetch request, allowing the simulator to monitor how many prefetches from each age group are successful—that is, how many result in a later cache hit. With this data, we calculated the accuracy of prefetches from each age group.

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

Overall, PC-indexed methods perform better and consume less bandwidth than G/DC methods. The GHB PC/DC method outperforms conventional stride prefetching (table PC/CS) by 7 percent and has approximately the same memory traffic (for a prefetch degree of 16). These results show that the PC/DC method can prefetch the same access patterns as constant-stride prefetching and gets additional performance from its ability to prefetch more-complex delta access patterns. Figure 3b shows that PC-indexed methods don't benefit as much as G/DC methods do from the GHB's ability to reduce stale data. For conventional stride prefetching, table entries are associated with a specific load instruction. Even if a table entry has not been accessed for a long time, it is less likely that a load instruction's behavior has changed since it was last accessed.

Collectively, the new GHB prefetching methods perform as well or better than their conventional counterparts. The first GHB method, GHB G/DC, shows a 20 percent IPC improvement over distance prefetching, while reducing memory traffic 90 percent. The second GHB method, GHB PC/DC, shows a 7 percent IPC improvement over stride prefetching and generates the same amount of memory traffic. In support of our performance results, GHB prefetching has been independently verified to out-perform other proposed prefetching mechanisms.[8]

Extensions of our research will consider applications of the GHB to next-generation memory technologies. If anything, the importance of prefetching will grow over the next decade as relative memory latencies continue to increase. At the same time, there will be significant increases in memory bandwidth, as next-generation memory technologies move to more sophisticated topologies and signaling techniques. Unfortunately, increases in memory bandwidth will have little direct effect on single thread performance. However, more advanced prefetching methods may provide promising indirect ways of using extra memory bandwidth to improve single thread performance. Looking beyond the GHB mechanism, our research makes another important contribution by illustrating that it is less important to produce prefetches quickly than it is to produce more accurate prefetches with higher prefetch coverage. This result supports future research into more advanced prefetching structures and algorithms that take longer, but produce better prefetches. MICRO

Acknowledgments

This research was funded by an Intel undergraduate research scholarship, a University of Wisconsin Hilldale undergraduate research fellowship, and by National Science Foundation grants CCR-0311361 and EIA-0071924.

References

1. A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Trans. Computers.*, vol. 11, no. 12, Dec. 1978, pp. 7-21.
2. J.M. Tendler et al., *POWER4 System Microarchitecture*, IBM tech. white paper, 2001.
3. T. Chen and J. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Trans. Computer Systems*, vol. 44, no. 5, May 1995, pp. 609-623.
4. D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *IEEE Trans. Computer Systems*, vol. 48, no. 2, 1999, pp. 121-133.
5. G.B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-Driven Study," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE Press, May 2002, pp. 195-206.
6. Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE Press, 2002, pp. 171-182.
7. D. Burger and T. Austin, The SimpleScalar Toolset, Version 3.0, http://www.simplescalar.org.
8. D. Gracia-Perez, G. Mouchard, and O. Temam, "MicroLib: A Case for Quantitative Comparison of Micro-Architecture Mechanisms," *Proc. 37th Int'l Symp. Microarchitecutre* (MICRO 04), IEEE Press, 2004, pp 43-54.

Kyle J. Nesbit is a PhD candidate in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include processor performance modeling, power-efficient processor design, virtual machines, and memory optimization. Nesbit has a BS in electrical engineering from the University of Wisconsin-Madison. He is a student member of the IEEE.

James E. Smith is a professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His current research interests include high-performance and power-efficient processor implementations, processor performance modeling, and virtual machines. Smith has a PhD in computer science from the University of Illinois. He is a member of the IEEE and ACM.

Direct questions and comments about this article to James E. Smith, Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison, 1415 Engineering Drive, Madison, WI 53706; es@ece.wisc.edu.