Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications

Yuan Chou
Architecture Technology Group
Microelectronics Division
Sun Microsystems
yuan.chou@sun.com

Abstract

The performance of many important commercial workloads, such as on-line transaction processing, is limited by the frequent stalls due to off-chip instruction and data accesses. These applications are characterized by irregular control flow and complex data access patterns that render many low-cost prefetching schemes, such as stream-based and stride-based prefetching, ineffective. For such applications, correlation-based prefetching, which is capable of capturing complex data access patterns, has been shown to be a more promising approach. However, the large instruction and data working sets of these applications require extremely large correlation tables, making these tables impractical to be implemented on-chip. This paper proposes the epoch-based correlation prefetcher, which cost-effectively stores its correlation table in main memory and exploits the concept of epochs to hide the long latency of its correlation table access, and which attempts to eliminate entire epochs instead of individual instruction and data misses. Experimental results demonstrate that the epoch-based correlation prefetcher, which requires minimal on-chip real estate to implement, improves the performance of a suite of important commercial benchmarks by 13% to 31% and significantly outperforms previously proposed correlation prefetchers.

1 Introduction

Commercial applications such as online transaction processing (OLTP), web servers and application servers represent important workloads for shared-memory multiprocessor servers. The performance of these applications is limited by the high cache miss rates resulting from their large instruction and data footprints [1-5]. In particular, a large fraction of their execution time is attributable to cache misses that result in off-chip memory accesses. For example, [5] found that, in today's database workloads, almost two-thirds of execution time is spent in these accesses. Instruction prefetching [6-10] and data prefetching [11-25], implemented either in hardware or software, are potential techniques for reducing the number of off-chip cache misses. In fact, many of today's processors implement some form of hardware stream-based data prefetching [11,12,21]. Unfortunately, the irregular control flow and complex data access patterns exhibited by many commercial applications render these simple and area efficient prefetchers ineffective. In contrast, correlation prefetching [18-24], which is able to remember complex recurring data access patterns, is a more promising approach for these applications. However, due to the large working set of commercial applications, the correlation table used to record the data access patterns must be very large in order for the approach to be effective.

It was observed by [26,27], with off-chip latencies of several hundred cycles, on-chip computation latencies separating overlappable off-chip accesses become insignificant such that these accesses essentially issue and complete at the same time. Moreover, very little computation can be accomplished while the accesses are outstanding. In such a situation, instruction execution tends to separate itself into epochs, which are recurring periods of on-chip computations followed by off-chip accesses. Removing all the off-chip accesses of an epoch eliminates that epoch, and reducing the number of epochs directly improves overall performance. This paper demonstrates how this concept of epochs can be exploited to implement a more effective correlation prefetcher that specifically targets the removal of epochs. This prefetcher does not waste predictor state on eliminating off-chip misses that are already overlapped with the triggering miss.

The biggest drawback of correlation prefetching is the large table that is required to store the correlations of miss addresses. For database workloads, the storage required to record the correlations for each executing thread can be in the order of multiple megabytes. With the industry's current trend towards aggressive chip multiprocessing, the aggregate table size needed to store the correlations for all threads executing on the processor chip is enormous. Moreover, implementing the tables on-chip consumes precious chip area that can be utilized by adding more processor cores. Therefore, storing the tables off-chip, such as in main memory, is highly attractive. This is especially true for server processors, whose systems are typically endowed with copious amounts of main memory. While the latency of accessing main memory may seem prohibitively high, this paper makes a key insight that this latency can be hidden by exploiting memory-level parallelism (MLP) [26,29]. In other words, the correlation table in main memory can be accessed while the processor is stalled on another memory or off-chip access. This epoch-based approach of hiding the latency of a table access under a previous epoch makes it feasible to store it in main memory. While reading and updating the table consumes memory bandwidth, the limited MLP in commercial applications [26] means that excess memory bandwidth is usually available, especially if the processor was designed to support bandwidth intensive high-performance com-

1072-4451/07 $25.00 © 2007 IEEE
DOI 10.1109/MICRO.2007.39

puting (HPC) workloads. For example, the new dual-core, dual-threaded IBM Power 6 processor [30] supports 300 GB/s of memory bandwidth. Assuming a memory latency of 200 ns, the bandwidth delay product of this processor is 300 GB/s * 200 ns = 6 KB. Assuming a L2 cache line size of 128 bytes, it takes more than 100 concurrent memory accesses by each of its four executing threads to fully saturate the available memory bandwidth. Other than satisfying demand requests and predictor table reads and updates, this abundant memory bandwidth can be utilized by the prefetches issued by the correlation prefetcher.

The rest of this paper is organized as follows. Section 2 describes prior related work as well as the concept of epochs. Section 3 presents the ideas of epoch-based optimizations and show how they are applied in the context of a correlation prefetcher. Section 4 describes the evaluation methodology employed while Section 5 presents the experimental evaluation of the epoch-based correlation prefetcher. Finally, Section 6 describes future work and concludes this paper.

2 Background and Related Work

This section familiarizes the reader with the concept of *epochs*, which was first proposed in [27], and then describes previous related work on correlation prefetching. It prepares the reader with the background to better understand the epoch-based correlation prefetcher presented in Section 3.

2.1 Epoch MLP Model

With off-chip latencies of several hundred cycles, on-chip computation[1] latencies separating overlappable off-chip accesses become increasingly insignificant relative to off-chip access latencies. In such a situation, illustrated in Figure 1, overlappable off-chip accesses appear to issue and complete at the same time, and instruction execution tends to separate itself into recurring periods of on-chip computations and off-chip accesses. In the epoch MLP model, each such period of on-chip computations followed by off-chip access(es) is called an *epoch*. More precisely, an epoch is a time slice of program execution starting from the end of the previous epoch through the first off-chip access and extending to the cycle when this first access has completed. Within an epoch, all overlappable off-chip accesses are assumed to issue and complete at the same time. The instruction that is responsible for the first off-chip access of an epoch is called the *epoch trigger*. The set of instructions from the processor's dynamic instruction stream (DIS) that can be executed in an epoch is called the *epoch set*. The entire DIS is partitioned into epoch sets such that all the instructions in epoch set *i* execute in the *i*th epoch.

A *window termination* condition is a condition that prevents the processor from executing any subsequent instructions in the DIS till all earlier off-chip accesses are resolved. Therefore, window termination conditions, together with data dependences, determine epoch sets and which off-chip misses can execute in an epoch. [26] found that the important window termination conditions are: issue window full, reorder buffer full, serializing instructions, mispredicted branches that are dependent on an off-chip miss, and off-chip instruction misses.

In a real processor (or a cycle-accurate simulator), epochs can be tracked by detecting epoch triggers. Recall that the epoch trigger is the first off-chip cache miss in an epoch. Accordingly, when the number of outstanding off-chip misses transitions from 0 to 1, the epoch count is incremented.

The number of epochs is related to overall processor performance in the following manner. The total execution time of an application, $Cycles_{total}$, is given by the following equation:

$$Cycles_{total} = Cycles_{on\text{-}chip}(1 - Overlap) + Cycles_{off\text{-}chip}$$

where $Cycles_{on\text{-}chip}$ is the number of cycles spent in on-chip computation, $Cycles_{off\text{-}chip}$ is the number of cycles spent in off-chip accesses, and Overlap is the fraction of on-chip cycles that were overlapped with off-chip cycles. $Cycles_{on\text{-}chip}$ can be measured on a cycle simulator by assuming that the furthest on-chip cache is perfect. In the epoch model, $Cycles_{off\text{-}chip}$ is the product of the number of epochs and the off-chip miss penalty. Therefore:

$$Cycles_{total} = Cycles_{perf}(1\text{-}Overlap) + (Epochs * MissPenalty)$$

Alternatively, the above equation can be expressed in terms of overall clocks per instructions (CPI):

$$CPI_{overall} = CPI_{perf}(1\text{-}Overlap) + (EPI * MissPenalty)$$

Reducing the number of epochs per instruction (EPI) directly reduces off-chip CPI. Moreover, since Overlap varies very little and can be approximated as a constant, EPI also has a linear relationship with overall CPI. Consequently, reducing EPI directly translates into improvements in overall CPI and overall performance.

2.2 Correlation Prefetching

The idea behind correlation prefetching is to use M previous miss addresses $P_1...P_M$ to predict N future miss addresses $F_1...F_N$. Typically, only the immediately previous miss address is used to predict the future miss addresses, i.e. M = 1 [20]. N is sometimes referred to as the prefetch depth [25] or the number of successors [24]. In the original correlation prefetching schemes [18-20], prefetch depth N = 1 and F_1 is the immediate successor of P_1 (i.e. F_1 is the next miss after P_1 in the dynamic miss stream). Note that these miss addresses can either belong to instruction fetches or data accesses.

[1] On-chip computation refers to data computations, address computations, as well as instruction fetches and loads that hit in the on-chip caches.

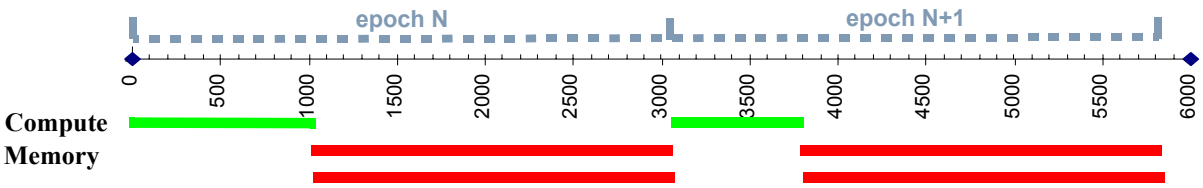

Figure 1. Example where off-chip latency is much greater than on-chip latencies.

In order to encode any correlation pattern, generalized correlation prefetching schemes maintain the correlations in a correlation table. Each table entry contains a tag to encode $P_1...P_M$ and it also contains prefetch addresses $F_1...F_N$.

Since it is possible that the miss address P_1 may be followed by one of several different miss addresses depending on the program control flow and/or data flow, Joseph and Grunwald [20] proposed Markov prefetching, which maintains and prefetches K miss addresses for F_1. K is sometimes also referred to as the prefetch width [25] or the number of rows [24]. They found that a prefetch width of K = 4 is a good trade-off.

A drawback of the original correlation prefetching schemes is the lack of timeliness. To address this, Nesbit and Smith [25] propose increasing prefetch depth N in addition to increasing prefetch width K. They found that this achieves better performance than increasing prefetch width or prefetch depth alone. Another innovation of this work is the use of a circular buffer, called the Global History Buffer (GHB), instead of a table to maintain the correlations.

The biggest disadvantage of correlation prefetching schemes is the large correlation table required. This is especially true for applications that have large working sets, such as commercial applications. To address this problem, Solihin et al [24] proposed storing the correlation table in main memory and implementing the prefetcher as a user-level thread running on a general purpose processor that resides either in the North Bridge chip or in the DRAM chip. Their proposal belongs to a category called memory-side prefetching, in which prefetching is initiated by an engine that resides close to main memory or inside of it. In addition, in order to improve timeliness, they proposed increasing prefetch depth N to 3. In their scheme, prefetch width K = 2.

Stream-based data prefetching [11,12,21] and next-line instruction prefetching [6] are examples of restricted correlation prefetching. In these schemes, each correlation is compactly encoded as one or more strides. To reduce the size of the correlation table in the generalized scheme, hybrid schemes that combine a stream-based prediction table with a correlation table have been proposed [21].

2.3 Other Related Work

While all the correlation prefetching work described above attempt to prefetch all types of misses, Wenisch et al's Tempral Streaming Engine (TSE) [16] exploits correlation to prefetch specifically for coherent read misses in a distributed shared memory multiprocessor. In their scheme, correlation information is stored in memory and directories, and the correlation among coherent read misses discovered by one processor is used by another processor for the latter's prefetching.

Lai et al [23] uses address correlation for generating prefetches to replace cache lines that are predicted to be dead. Ferdman and Falsafi [41] extends this work to store the correlations off-chip and bring them into an on-chip structure when needed. Instead of exploiting correlation among miss addresses, Hu et al's [15] Tag Correlating Prefetching (TCP) scheme exploits correlations among cache tags (i.e. the high order bits of miss addresses). In addition, schemes to exploit spatial correlation [28, 36, 39, 40] have also been proposed. For example, Spatial Memory Streaming [36] exploits spatial correlation to stream prefetch requests for lines from a 2KB memory region when the first access to the region is detected.

Hardware prefetching is a mature field of research and many hardware prefetching schemes have been proposed over the years. While it is not possible to exhaustively list and compare all these schemes, Perez et al [37] experimentally compared some of the recent well known schemes.

Lastly, the concept of epochs is similar to the concept of *eras* described in [31], and the term memory-level parallelism (MLP) was first used in [29].

3 Epoch-Based Optimizations

This section shows how the epoch concept leads to key insights that not only enable the design of a more effective correlation prefetcher but also makes it possible to store the large correlation table in main memory and still access it in a timely manner. This section then compares and contrasts the resulting epoch-based correlation prefetcher with previously proposed correlation prefetchers. This is followed by a description of the implementation details of the epoch-based correlation prefetcher.

3.1 Epoch-Based Correlation Prefetching

The first key insight observed from the epoch model is that *eliminating entire epochs directly improves overall performance*. Therefore, a prefetcher should attempt to eliminate entire epochs by prefetching every off-chip miss that belongs to an epoch. Traditional correlation prefetchers target individual misses and eliminate epochs as a by-product of eliminating individual misses. However, eliminating individual misses does not always result in a removal of an epoch and therefore may not always improve performance. Moreover, correlation prefetchers may inadvertently try to eliminate misses that are naturally overlapped by the processor's out-of-order instruction issue.

Based on this insight, the epoch-based correlation prefetcher (EBCP) is proposed. In the epoch-based correlation prefetcher, the first miss address in an epoch is used to predict the addresses of *all* the misses in the next X epochs. Experimental data indicates that X = 2 is a good trade-off between coverage and accuracy, so X = 2 is assumed for the rest of this paper. Note that the address of the other misses in the current epoch is not predicted as these prefetches will not be timely. This helps to reduce the amount of storage required by the correlation table.

Consider the following example sequence of miss addresses A, B, C, D, E, F, G, H and I. This sequence is assumed to recur after a sufficiently long period of time so that all their associated cache lines have been evicted from the L2 cache. Also suppose that, in the absence of prefetching, misses A and B issue in epoch i, misses C, D and E issue in epoch i+1, misses F and G issue in epoch i+2, and misses H and I issue in epoch i+3.

Epoch	i	i+1	i+2	i+3
Miss Addresses	A, B	C, D, E	F, G	H, I

In the traditional correlation prefetching schemes (such as the Global History Buffer depth correlation prefetching scheme [25]), assuming a prefetch depth of two, prefetch addresses B and C are issued in the epoch i, averting miss C

but not B (because it is not timely). Prefetch addresses E and F are issued in epoch i+1, averting miss F but not E. Prefetch addresses H and I are issued in epoch i+2 and avert misses H and I. It takes three epochs to process all the misses, eliminating one epoch compared to no prefetching.

Traditional correlation prefetcher

Epoch	i	i+1	i+2	i+3
Miss Addresses	A, B	C, D, E	G	-
Prefetches Issued	B, C	E, F	H, I	-

In contrast, in the epoch-based correlation prefetching scheme, assuming prefetching the next two epochs (i.e. X = 2), prefetch addresses C, D, E, F and G are issued in epoch i because all the prefetch addresses in epoch i+1 and epoch i+2 are prefetched. Miss H and miss I are not prefetched since they are outside the next two epochs. The timely issue of the prefetches for C, D, E, F and G averts these misses, and the end result is that it only takes two epochs to process all the misses.

Epoch-based scheme

Epoch	i	i+1	i+2	i+3
Miss Addresses	A, B	H, I	-	-
Prefetches Issued	C, D, E, F, G	-	-	-

In the above discussion of the traditional correlation prefetcher and the epoch-based correlation prefetcher, it is assumed that the correlation table is stored on-chip so that the prefetches can be issued in the same epoch that the correlation table is read. Unfortunately, for correlation prefetchers to work effectively for commercial applications that have immense instruction and data working sets, very large correlation tables are required. The next section shows how, by exploiting another insight from the epoch model, the epoch-based correlation prefetcher can store its correlation table in main memory and yet access it in a timely manner.

3.2 Timely Access of Main Memory Predictor Tables

The second key insight observed from the epoch model is that *it is possible to hide the latency of accessing a slow predictor under a prior off-chip miss*. In other words, if the prediction is needed in the current epoch, the predictor access can be timely as long as the access is made in an earlier epoch. In the context of the epoch-based correlation prefetcher, the prefetcher can be modified so that the miss address of epoch i is used to prefetch for the misses in epoch i+2 and epoch i+3 (rather than for the misses in epoch i+1 and epoch i+2). The predictor can be read in epoch i and the prefetches can be issued and completed in epoch i+1. This makes it feasible to store the large correlation table off-chip, such as in main memory, and still issue the prefetches in a timely manner.

To illustrate how this works, consider the previous example sequence of miss addresses A, B, C, D, E, F, G, H and I. As mentioned previously, in the absence of prefetching, misses A and B issue in epoch i, misses C, D and E issue in epoch i+1, misses F and G issue in epoch i+2, and misses H and I issue in epoch i+3.

In the epoch-based correlation prefetching scheme with the correlation table located in main memory, prefetch addresses F, G, H and I are read from the main memory correlation table in epoch i and the prefetches are issued in epoch i+1. This averts misses F, G, H and I, and consequently, it only takes two epochs to process all the misses.

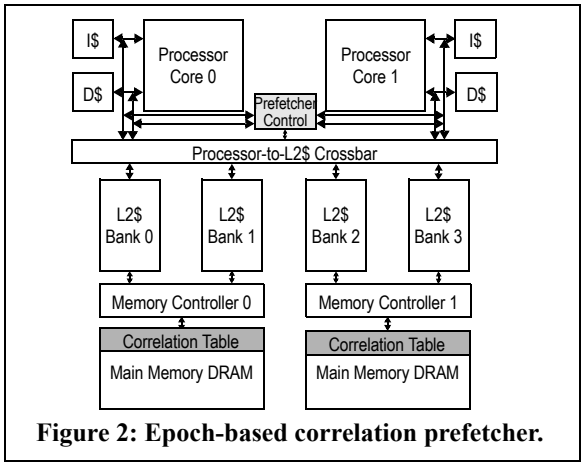

Figure 2: Epoch-based correlation prefetcher.

Epoch-based scheme (main memory correlation table)

Epoch	i	i+1	i+2	i+3
Miss Addresses	A, B	C, D, E	-	-
Prefetches Issued		F, G, H, I	-	-

It is useful to note that the epoch-based correlation works best for workloads with low or medium memory-level parallelism (MLP) [26]. Compared to traditional correlation prefetchers, it is especially effective for medium MLP workloads. It does not work well for high MLP workloads since it is difficult to detect epoch boundaries for these workloads. However, naturally high MLP workloads are least likely to benefit from a prefetcher anyway, so this is not a significant drawback.

Figure 2 depicts the epoch-based correlation prefetcher as it is implemented in a multi-core processor with a shared L2 cache that is banked. Even though the prefetcher only prefetches L2 cache misses, the prefetcher control is located in front of the core-to-L2 cache crossbar rather than behind it. The prefetcher control tracks L1 miss requests that are sent by the processor cores to the L2 cache banks. The L2 cache banks inform the prefetcher control which of these requests also miss the L2 cache. This arrangement allows the prefetcher control to see the entire L2 miss stream of every thread executing on the processor. This is crucial since, for an executing thread, the correlation among misses seen by one particular L2 bank is much weaker than the correlation among the thread's entire miss stream. Note that, like the L2 cache, the correlation table can possibly be shared by all the executing threads on the processor.

3.3 Comparison with Other Correlation Prefetchers

In this section, the epoch-based correlation prefetcher is compared and contrasted with previously proposed correlation prefetchers that are conceptually closest to it.

3.3.1 Comparison with Solihin's Prefetcher

As described earlier in Section 2.2, Solihin's memory-side correlation prefetching scheme [24] attempts to prefetch the next three miss addresses after the current miss. For the example miss sequence, Solihin's scheme reads prefetch

addresses B, C, D and E (B, C, and D are generated by miss A while C, D and E are generated by miss B) from the memory-based correlation table in epoch i and issues the prefetches in the epoch i+1. Unfortunately, these prefetches are not timely. Prefetch addresses F, G and H are read from the correlation table in epoch i+1 and issued in epoch i+2, averting only miss H. Prefetch address I is read from the correlation table in epoch i+2 and issued in epoch i+3 and is therefore not timely. As a result, four epochs are still required in order to process all the misses. Note that in this description of Solihin's scheme, it is assumed that the latency of accessing the memory chips is considerably greater than the latency of reaching the North Bridge chip from the processor. This assumption is reasonable for new memory technologies such as FBDIMMs [32] which require multiple chip crossings when the memory sockets are fully populated (as they are likely to be in server processor systems).

Solihin's scheme

Epoch	i	i+1	i+2	i+3
Miss Addresses	A, B	C, D, E	F, G	I
Prefetches Issued	-	B, C, D, E	F, G, H	-

While the epoch-based correlation prefetcher and Solihin's prefetch both store the correlation table in main memory, the epoch-based scheme locates the predictor control on-chip (in front of the processor-to-L2 cache crossbar) while Solihin's scheme locates it (in the form of a second processor) either on the North Bridge chip or embeds it within the DRAM chip. Solihin's scheme is less effective when employed in multi-core/multi-threaded processors where the requests received by the memory controller is an interleaving of requests from the different threads executing concurrently on the processor. Such interleaved request streams do not exhibit sufficient correlation to enable effective prefetching. Furthermore, multi-core processors generally have multiple memory controllers. Any one memory controller receives a subset of the memory requests from a particular core. Locating the predictor control in the memory controller prevents the prefetcher from seeing the entire miss stream of each executing thread. Therefore, Solihin's scheme will not work for processors with more than one memory controller. Since the epoch-based scheme locates the predictor control on-chip and in front of the processor-to-L2 cache crossbar, it is immune to such problems.

Moreover, unlike Solihin's scheme, the epoch-based correlation prefetcher requires little or no modification to the memory controller, and it can take advantage of industry-standard memory technology such as FBDIMMs and DDR2/3. Solihin's scheme is also more expensive in that it requires a second general-purpose processor that resides either in the North Bridge chip or in the DRAM chip.

3.3.2 Comparison with Temporal Streaming Engine

Like the epoch-based correlation prefetcher, the Temporal Streaming Engine (TSE) [16] also stores its correlation table in main memory. However, it requires three hops on the system interconnect to obtain the correlation table entry. The additional hop is needed because the directory must first be consulted to determine which processor node's main memory holds the correlation table entry. In contrast, the epoch-based correlation prefetcher only requires two hops on the system interconnect to obtain the correlation table entry. The TSE is also much more complex than the epoch-based correlation prefetcher. Its requires a stream engine that locates, compares and follows multiple streams, whereas the epoch-based correlation prefetcher simply sends out prefetches for all the prefetch addresses in a correlation table entry.

While the epoch-based scheme explicitly attempts to fully hide the latency of accessing the main memory correlation table, no such attempt is made in the TSE. Instead, the TSE relies on the cost of the main memory correlation table access to be amortized across the stream of prefetches generated. Moreover, similar to other conventional correlation prefetchers, the TSE does not explicitly attempt to eliminate epochs.

Most importantly, while the epoch-based correlation prefetcher targets all types of L2 cache misses, the Temporal Streaming Engine (TSE) [16] only works for reducing coherent read misses. The current industry trend towards chip-multiprocessors with many cores on a single processor chip sharing an L2 cache and fewer processor chips in the system means that coherent read misses will be an increasingly small fraction of all L2 cache misses in future systems. For such future systems, it is essential to target not only coherence read misses but also all types of L2 cache misses.

3.4 Implementation Details

In this section, details of the epoch-based correlation prefetcher is described. A description of how the main memory for the correlation table is allocated and de-allocated is followed by descriptions of how the prefetcher is trained and how the prefetcher generates prefetches.

3.4.1 Main Memory Correlation Table Allocation

On start-up, the on-chip prefetcher control requests the operating system for a region of memory to store the correlation table. The request can be made by causing a trap, thereby transferring control to the operating system. In a possible implementation, the operating system allocates a series of contiguous physical pages and returns the base physical address of the first page as well as the size of the allocated region (i.e. the aggregate size of all the allocated pages). This base physical address is the base memory address of the correlation table. The on-chip predictor control operates on physical addresses, obviating the need for address translations in accessing and updating the predictor table. On a successful memory allocation, the prefetcher enters the active state, where it learns correlations in the miss address stream, updates the main memory correlation table and issues prefetches based on the information stored in the table.

If the operating system runs out of physical memory, it may reclaim the physical memory region used by the correlation table. In response, the on-chip prefetcher control enters the inactive state. Subsequently, after a pre-determined time period, the on-chip prefetcher control can attempt to request the operating system for another physical memory region to store the correlation table. If the attempt is successful, the prefetcher re-enters the active state.

3.4.2 Prefetcher Learning

To record the physical addresses of the L2 cache misses

in each epoch, a structure named the Epoch Miss Addresses Buffer (EMAB) is employed. The EMAB is a circular buffer with four entries, with each entry containing the miss addresses of one epoch. The first entry contains the miss addresses of the current epoch, and the other three entries contain the miss addresses of the three previous epochs. Since the EMAB is a circular buffer, the oldest entry is overwritten when a new entry is needed.

As L2 cache instruction and load misses are encountered by the processor, their physical addresses are recorded in the current epoch's EMAB entry. Because the assumed baseline processor model implements the weak consistency memory model, store prefetching is not essential [27] and therefore, L2 cache store misses are not recorded in the EMAB entry. As mentioned previously, epochs are tracked by incrementing the epoch count when the number of outstanding L2 cache misses transitions from 0 to 1. An increment of the epoch count signifies that the current epoch has ended and a new epoch is beginning. At this juncture, the oldest entry of the EMAB is inspected. The first miss address of the entry is used as the key to index the correlation table.

As shown in Figure 3, each correlation table entry contains a tag, a number of prefetch addresses (the exact number of prefetch addresses being dependent on the prefetch degree and is evaluated in Section 5) as well as LRU information. To reduce the memory bandwidth needed to access the table, it is direct-mapped and the size of each entry should fit within the natural unit of data transfer to and from memory (which is usually the line size of the last level of cache and is 64B in the default processor configuration). Assuming that each prefetch address can be compressed by only storing the lower bytes and using the upper bytes from the tag, eight prefetch address can easily be accommodated within the 64B L2 cache line.

Tag	LRU info	Prefetch Address 0	.....	Prefetch Address N
....				
....				

Figure 3: Correlation table organization.

To access the correlation table, the on-chip prefetcher control adds the correlation table index to the base memory addresses of the correlation table and the resulting physical address is used to generate a request to read main memory. These requests bypass the cache hierarchy and go directly to main memory. In order to avoid delaying memory requests generated by demand accesses, these requests are designated a lower priority. They will only be serviced when there are no demand accesses outstanding. If there is a match in the correlation table, the miss addresses in the other EMAB entries are used to update the correlation table entry. Specifically, the miss addresses contained in the two latest EMAB entries are used. The correlation table entry is updated by replacing some or all of the existing prefetch addresses of the entry with the miss addresses from the EMAB. The least recently used (LRU) replacement policy is used. If there is no match in the correlation table, a new correlation table entry is allocated by overwriting the existing correlation table entry of the same index, and the miss addresses from the EMAB are copied to the prefetch addresses of the correlation table entry. If there are more miss addresses from the EMAB than can be accommodated in the correlation table entry, priority is given to the miss addresses from the older of the two epochs, since these addresses are more likely to generate accurate prefetches.

To illustrate how the correlation table update works, consider again the example from Section 3.1.

Epoch	i	i+1	i+2	i+3
Miss Addresses	A, B	C, D, E	F, G	H, I

For discussion purposes, assume that the current epoch which has just ended is epoch i+3. The EMAB's four entries contain the miss addresses of epochs i, i+1, i+2 and i+3. The first miss address of epoch i's EMAB entry is used to index the correlation table and the miss addresses of epoch i+2 and i+3 are copied into the correlation table entry.

While the above implementation only uses the first miss in the oldest epoch's EMAB entry to insert/update a correlation table entry, an alternative implementation may use all the misses in the oldest epoch's EMAB entry to insert/update correlation table entries. However, simulations showed that the latter implementation requires larger correlation tables and only improves performance marginally, so it is not discussed further in this paper.

3.4.3 Prefetch Address Generation

When the first L2 instruction or load miss (or prefetch buffer hit) in a new epoch is encountered, its physical address (or physical PC in the case of an instruction miss) is used to generate the correlation table index to look up the correlation table. The on-chip prefetcher control adds the correlation table index to the base memory addresses of the correlation table and the resulting physical address is used to generate a request to read the correlation table entry from main memory. The prefetch addresses of the matching correlation table entry is then used to generate prefetch requests. These prefetch requests are designated as lower priority than demand accesses, so that they never delay demand accesses and are only serviced where there is unused memory bandwidth. The prefetch requests bring instruction/data into a prefetch buffer that is searched in parallel with the L2 cache.

Subsequent L2 instruction and load misses encountered in the epoch do not look up the correlation table. Simulations (not shown here due to space constraints) showed that allowing them to look up the correlation table and generate prefetches results in little performance improvement. To update the LRU information of the correlation table entry, each prefetch buffer entry also contains the index of the correlation table entry that generated the prefetch. A hit in the prefetch buffer causes the LRU information of the corresponding correlation table entry to be updated.

3.4.4 Memory Bandwidth Usage

Apart from the actual prefetch requests, the epoch-based correlation prefetcher requires two additional memory read requests and two additional memory write requests as a consequence of its correlation table being stored in main memory. The first read request is needed to read the correlation table to obtain the prefetch addresses and the second read request is

needed to read the correlation table in preparation for updating it with the latest information from the EMAB. The first write request is used to update the correlation table with the latest information from the EMAB and the second write request is used to update the LRU information in the correlation table entry on a prefetch buffer hit. Note that other than the first read request, the other memory requests are not timing critical. They can be done whenever there is spare memory bandwidth and are therefore assigned the lowest priority. Also note that the two additional read requests are negligible compared to the eight that can be issued in the tuned version of the prefetcher (please see Section 5). Furthermore, the experimental results in Section 5 demonstrate that the default processor configuration has sufficient memory bandwidth to sustain these additional memory requests.

4 Experimental Methodology

4.1 Performance Metrics

The primary performance metric used to evaluate the epoch-based correlation prefetcher is the improvement in overall performance relative to no prefetching. Overall performance is determined by measuring overall cycles per instruction (CPI) using a cycle-accurate timing simulator. Secondary metrics that are also employed include coverage, accuracy and the reduction in epochs per instruction (EPI).

4.2 Benchmarks

Four commercial workloads are used in this study: a database workload, TPC-W [34], SPECjbb2005 [35], and SPECjAppServer2004 [35]. The database workload is a large-scale online transaction processing set-up with a very large database that is stored on hundreds of disks. TPC-W is a transactional web benchmark that simulates the activities of a business oriented transactional web server. SPECjbb2005 is a server-side Java benchmark that emphasizes business logic and object manipulation, the middle tier of a 3-tier system. SPECjAppServer2004 (Java Application Server) is a client/server benchmark for measuring the performance of Java Enterprise Application Servers using a subset of J2EE APIs in a complete end-to-end web application.

The SPARC binaries used to generate the traces are highly optimized. For example, aggressive post-link code layout optimizations have been performed to minimize the number of instruction cache misses. The traces were generated by a full-system simulator and contain both kernel and user instructions. The traces were collected when the workloads were warmed and running in steady state and they were meticulously validated against hardware counter statistics. These traces contain register and memory values that allow the re-creation of processor and memory state during cycle-accurate processor simulation.

In all the experiments, the first 150M instructions in the trace are used to warm the caches and the prefetchers and the next 100M instructions are used to collect statistics. All four workloads are transaction-oriented and do not exhibit phase changes. Therefore, 100M instructions are sufficient for collecting a representative transaction mix that enables accurate statistics collection.

4.3 Cycle-Accurate Timing Simulator

This is a simulator that was designed to explore microarchitecture ideas for future commercial SPARC [33] processors. The simulator is capable of modeling the many aspects of the processor pipeline in detail. It also contains a memory interconnect model that accurately accounts for bandwidth constraints. While the simulator is trace-driven, it is augmented with an instruction set emulator. Using the register and memory values from the traces, the emulator allows the simulator to accurately model control and data speculation, thereby achieving the accuracy of execution-driven simulation for workloads with well behaved locking behavior, as demonstrated in [38]. All the benchmarks used in this study are properly tuned and exhibit well-behaved locking behavior.

4.4 Default Processor Configuration

The following default processor configuration was used in the experimental evaluations. The simulator was configured so that prefetches and correlation table reads and updates are always designated a lower priority than demand accesses. This ensures that demand accesses are not delayed by prefetches or correlation table accesses.

Core frequency	3 GHz
Number of cores	1
Front-end	Fetch up to 4 instructions per cycle, 32 entry fetch buffer
Branch prediction	64K entry gshare, 4K entry BTB, 16 entry RAS
Decode	Decode and rename up to 4 instructions per cycle
Issue queue/ ROB	64 entry issue queue, 128 entry reorder buffer
Issue policy	Out-of-order, loads issue OOO wrt other loads and stores and speculate past earlier unresolved stores, branches issue OOO wrt other branches
Function units	2 ALUs, 1 load/store unit, 1 branch unit, 1 FP multiply unit, 1 FP add unit
Store handling	32 entry store buffer, 32 entry store queue, 8 byte store coalescing
Load handling	64 entry load buffer
Memory model	Weak consistency
Retire	Retire up to 4 instructions per cycle
TLB	64 entry fully associate ITLB, 64 entry fully associative DTLB, 2K entry shared L2 TLB
L1 cache (I+D)	32KB 4-way set associative, LRU replacement, 64B lines, 3 cycle hit latency
L1 MSHRs	32
L2 cache	2MB 4-way set-associative, LRU replacement, 64B lines, 20 cycle hit latency
L2 MSHRs	32
L3 cache	None
Main memory	500 cycle latency (unloaded)
Interconnect	600 MHz, 16B wide read bus, 8B wide write bus, split transaction, 9.6 GB/s read bandwidth, 4.8 GB/s write bandwidth

5 Experimental Results

This section is organized as follows. The baseline (no prefetching) results for the benchmarks are shown in Section 5.1. The design space exploration of the epoch-based correlation prefetcher is presented in Section 5.2, while the results of an experiment showing the sensitivity of the prefetcher's performance to available memory bandwidth is shown in Section 5.3. This is followed by a performance comparison of the epoch-based correlation prefetcher with other recently proposed prefetching schemes in Section 5.4.

5.1 Baseline Results

Table 1 shows the $CPI_{overall}$ (equivalent to overall performance), epochs per instruction (EPI) and L2 cache miss rates (in misses per 1000 retired instructions) for the four benchmarks when run on the baseline processor without prefetching.

	Database	TPC-W	SPECjbb 2005	SPECjApp Server2004
CPI_{overal}	3.27	2.00	2.06	2.78
Epochs per 1000 insts	4.07	1.59	2.65	3.25
L2$ inst miss rate	1.00	0.71	0.12	1.57
L2$ load miss rate	6.23	1.27	4.30	2.64

Table 1: Statistics for baseline processor without correlation prefetching.

5.2 Design Space Exploration

To explore the predictor design space, an idealized predictor is initially assumed and then each design parameter is incrementally scaled back to achieve a realistic and balanced design configuration. The idealized predictor has an eight million entry correlation table, with each entry holding 32 prefetch addresses. It can issue a maximum of 32 prefetches per correlation table match and the prefetched lines are stored in a 1024 entry prefetch buffer. Prefetched lines are only copied from the prefetch buffer to the regular caches if they are actually used to satisfy a demand request.

5.2.1 Effect of Prefetch Bandwidth

Since memory bandwidth is an important resource, the first design parameter evaluated is the maximum number of prefetches that can be issued on a correlation table match (i.e. the prefetch degree). Figure 4 shows the effect of this design parameter on the improvement in overall performance and Figure 5 shows its effect on the reduction in EPI and L2 miss rates, as well as on the coverage and accuracy of the prefetcher. Note that as the prefetch degree is increased, prefetches may sometimes be dropped when the available memory bandwidth is saturated.

The results show that, for the memory read bandwidth of 9.6 GB/s and memory write bandwidth of 4.8 GB/s available in the default processor configuration, overall performance, EPI, as well as coverage continues to improve for all four benchmarks as the prefetch degree is increased. With a prefetch degree of 32, the epoch-based correlation prefetcher improves overall performance by 34% for the database OLTP workload, by 19% for TPC-W, by 43% for SPECjbb2005 and by 38% for SPECjAppServer2004. These are impressive performance gains for a prefetcher that requires no dedicated on-chip storage for its correlation table, and all the more so since the gains are obtained on workloads that are known to be challenging for prefetchers. For all four benchmarks, the reduction in EPI (as well as the improvement in overall performance) tracks very well with the improvement in coverage. This indicates that the prefetcher is successfully removing epochs with the misses it eliminates. This is in contrast to the Spatial Memory Streaming prefetcher [36], whose authors report a 40+% coverage on OLTP workloads but a performance improvement of only 5-10%. The authors suggest that this is because the Spatial Memory Streaming prefetcher, while successful in removing misses, is much less successful in removing epochs (at least on OLTP workloads).

Figure 5 shows that L2 load misses far outnumber L2 instruction misses for the database workload and SPECjbb2005. However, for TPC-W and SPECjAppServer04, L2 instruction misses constitute a significant fraction of all L2 misses. Except for TPC-W, the epoch-based correlation prefetcher is highly effective in reducing both L2 load misses as well as L2 instruction misses. For TPC-W, the prefetcher is more effective in reducing L2 instruction misses than in reducing L2 load misses.

As expected, the accuracy of the prefetcher decreases as the prefetch degree is increased. While the accuracy of the prefetcher is fairly low, the useless prefetches do not negatively impact performance since prefetches do not delay demand accesses. Nonetheless, since useless prefetches do waste power and it is also desirable to limit the number of prefetch addresses in a correlation table entry so that they can fit within the unit of memory transfer, the prefetch degree is limited to eight for the rest of the design space exploration. Clearly, higher performance can be obtained by increasing the number of prefetches, albeit at the expense of increased power consumption.

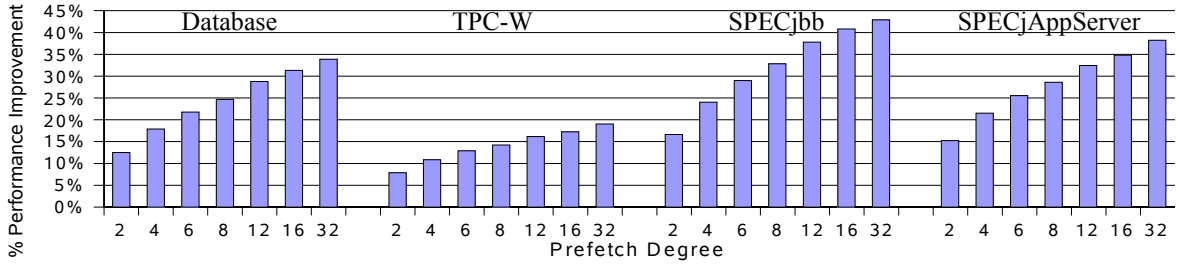

Figure 4: Effect of limiting number of prefetches on overall performance improvement.

Figure 5: Effect of limiting number of prefetches on EPI, L2 miss rate, coverage and accuracy.

5.2.2 Effect of Correlation Table Size

The size of the correlation table is a function of the number of table entries and the size of each correlation table entry. Since the correlation table is located in main memory, the size of each entry is constrained to be a multiple of the memory transfer unit size (which is 64B in the default processor configuration). As mentioned previously in Section 3.4.2, eight prefetch addresses (required to support the chosen prefetch degree of eight) should fit easily within 64B.

Figure 6 shows the effect of limiting the number of correlation table entries. For all four benchmarks, one million entries is sufficient to prevent significant performance erosion. Since the correlation table is located in main memory, one million entries (which corresponds to 64MB of memory) is feasible, especially for server processors which are usually endowed with generous amounts of main memory. It is also worth mentioning that, if desired, the number of correlation table entries can be specifically tailored to each application, since the table is a software structure residing in main memory and not a hardware structure.

5.2.3 Effect of Prefetch Buffer Size

The last design parameter to be varied is the number of prefetch buffer entries. The prefetch buffer is organized as a 4-way set associative structure. Figure 7 shows that a prefetch buffer with 64 entries, which requires only 512B of storage, is adequate.

Relative to no prefetching, this tuned epoch-based correlation prefetcher achieves an overall performance improvement of 23% on the database workload, 31% on SPECjbb2005, 13% on TPC-W, and 26% on SPECjAppServer2004. It is worth emphasizing again that these improvements are obtained without the need for any on-

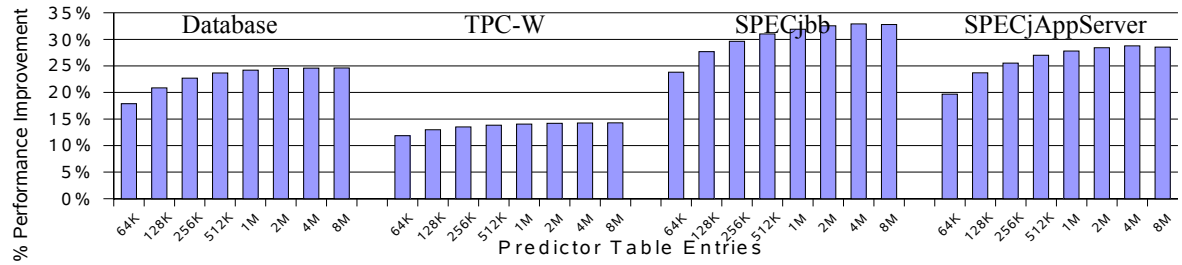

Figure 6: Effect of limiting number of predictor table entries on overall performance improvement.

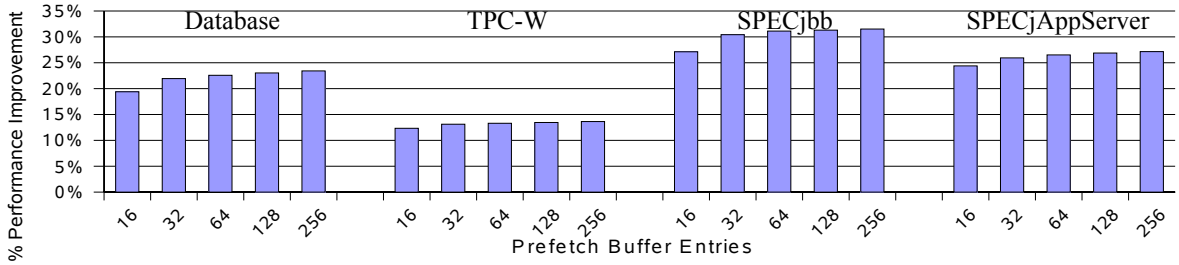

Figure 7: Effect of limiting number of prefetch buffer entries on overall performance improvement.

chip correlation table storage. The only on-chip structures are the EMAB, the small prefetch buffer and the simple prefetcher control, all of which are off the critical path and require minimal chip area to implement.

5.2.4 Sensitivity to Available Memory Bandwidth

Figure 8 shows the sensitivity of the epoch-based correlation prefetcher performance to available memory bandwidth. The dotted lines show the runs with 3.2GB/s of read bandwidth and 1.6 GB/s of write bandwidth. The light solid lines show the runs with 6.4 GB/s of read bandwidth and 3.2 GB/s of write bandwidth, while the dark solid lines show the runs with 9.6 GB/s of read bandwidth and 4.8 GB/s of write bandwidth (the default). The performance improvements reported are relative to the baseline processor without prefetching. The results show that the optimal prefetch degree depends on the benchmark and the available memory bandwidth. With 9.6 GB/s of available read bandwidth, performance continues to improve for all benchmarks as the prefetch degree is increased. With 6.4 GB/s of available read bandwidth, this is no longer true for the database workload or for SPECjbb2005. For these benchmarks, maximum performance is obtained with a prefetch degree of 16. With 3.2 GB/s of available read bandwidth, the performance of the database degree always declines as prefetch degree is increased. For SPECjbb2005 and SPECjAppServer2004, performance declines when prefetch degree is increased beyond eight.

The results clearly demonstrate that the performance of the epoch-based correlation prefetcher is sensitive to available memory bandwidth. Fortunately, as noted earlier, new server processors are endowed with tremendous amounts of memory bandwidth. As an example, the newly announced dual-core, dual-threaded IBM Power 6 processor [30] supports 300 GB/s of memory bandwidth. For such processors, it is feasible to trade-off increased memory bandwidth usage for the ability to store the correlation table in main memory.

5.3 Comparison with Other Prefetchers

The epoch-based correlation prefetcher is now compared with several well known and recently proposed prefetchers. Perez et al [37] experimentally compared twelve of the recent and/or well known prefetching schemes using the SPECcpu2000 benchmarks, and found that the Global History Buffer (GHB) with its PC/DC (program counter/ delta correlation) variant [25] scheme works the best among the twelve prefetchers. Therefore, the GHB-PC/DC scheme, which is essentially an address-based correlation prefetcher, was selected as one of the comparison points. In particular, two dif-

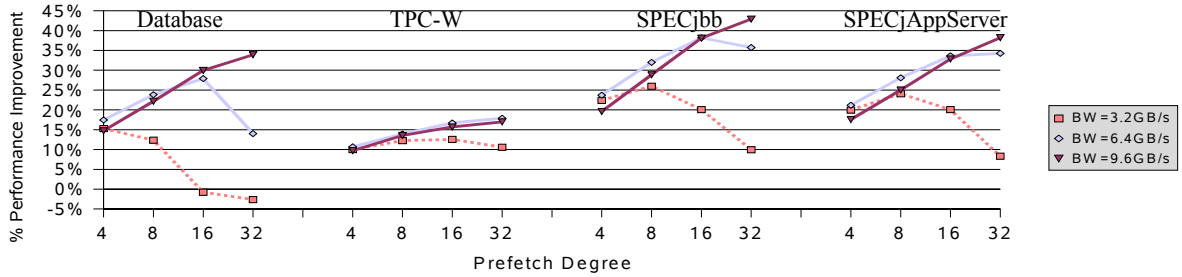

Figure 8: Effect of available memory bandwidth on epoch-based correlation prefetcher performance.

ferent table configurations of the GHB-PC/DC were simulated. The first configuration, termed *GHB small*, comprises of a 16K entry index table and a 16K entry global history buffer, and is estimated to require 256KB of storage. The second configuration, termed *GHB large*, comprises of a 256K entry index table and a 256K entry global history buffer, and is estimated to use 4MB of storage. The prefetch degree was set at 6 and a depth prefetching scheme was employed. A uniform prefetch of degree of 6 was chosen for all the prefetchers (except the Spatial Memory Streaming prefetcher) to ensure a fairer comparison.

The second prefetching scheme chosen is the Tag Correlating Prefetcher (TCP) [15]. This scheme exploits the correlation among tags rather than the correlation among addresses, which is supposed to reduce the size of the correlation table required. Again, two configurations of the TCP were simulated. The first configuration, termed *TCP small*, comprises of 2048 PHT sets, with 16 ways per PHT set. With the organization of L1 caches in the simulated processor configuration and assuming a 45-bit physical address, this configuration requires approximately 256KB of storage. The second configuration, termed *TCP large*, comprises of 32K PHT sets, 16 ways per PHT set, and requires approximately 4MB of storage. For both configurations, the THT contains 128 entries, matching the same number of sets in the L1 caches.

The third prefetching scheme selected is a stream prefetcher, and it was chosen because a stream prefetcher is implemented in many current high performance general purpose processors, such as the IBM Power 5 processor, the Fujitsu SPARC64-VI processor, the AMD Opteron processor and the Intel Pentium 4 processor. This stream prefetcher is capable of tracking up to 32 streams and handles, positive, negative and non-unit strides. On the detection and confirmation of a stream, it issues 6 prefetch requests and then attempts to keep 6 strides ahead of the request stream. Stream prefetchers are relatively inexpensive to implement because they require little storage.

The fourth prefetching scheme selected is the Spatial Memory Streaming (SMS) prefetcher [36], termed *SMS*. The spatial region size is 2KB, and the Accumulation Table and Filter Table are combined and contain 128 entries. The Pattern History Table (PHT), which is stored on-chip, is 16-way set-associative and contains a total of 16K entries. Since the line size of the data cache in the default processor configuration is 64B and the spatial region size is 2KB, the spatial pattern stored in each PHT entry can be encoded using 32 bits. Assuming that the PHT tag can be compressed into four bytes, the storage needed for the PHT is 128KB. Unlike the other prefetchers selected for this comparison, up to 32 prefetches (i.e. all lines in a spatial region) can be selected on a PHT match.

The fifth prefetching scheme chosen is Solihin's scheme [24], which is the scheme conceptually closest to the epoch-based correlation prefetcher, since it too stores its correlation table in main memory. In the original implementation [24], the prefetch depth is set at three and the prefetch width at two, with a maximum number of six prefetches being issued per correlation table entry match. This scheme is termed *Solihin 3,2*. Since [16] demonstrated the effectiveness of increasing prefetch depth, an enhanced version of Solihin's scheme with a prefetch depth of six and prefetch width of one is also included as another comparison point. This enhanced scheme is termed *Solihin 6,1*. Like the epoch-based correlation prefetcher, the main memory correlation table in *Solihin 3,2 and Solihin 6,1* contains one million entries. Therefore, both *Solihin 3,2 and Solihin 6,1* use the same amount of (main-memory) storage as the epoch-based correlation prefetcher.

To ensure a fair comparison with the other prefetchers, the epoch-based correlation prefetcher, labeled *EBCP*, is configured with a prefetch degree of six, and stores up to six prefetch addresses in its correlation table entry. Its main memory correlation table contains one million entries.

All the prefetchers in the comparison bring their prefetched lines into a 64 entry prefetch buffer. These lines are copied to the regular caches only when they are used to satisfy a demand request. Although the original Spatial Memory Streaming prefetcher [36] prefetches lines directly into the data cache, better performance was obtained for this prefetcher by bringing the prefetched lines into a prefetch buffer so this is the implementation used in the comparison. The Global History Buffer, Solihin and epoch-based correlation prefetchers specifically target L2 cache misses, while the other prefetchers attempt to prefetch L1 cache misses as well. Also, the Tag Correlation Prefetcher, stream prefetcher and the Spatial Memory Streaming prefetcher only target load misses while the other prefetchers also attempt to prefetch for instruction misses.

The results of the comparison is shown in Figure 9. The metric used is the overall performance improvement relative to the baseline processor without prefetching.

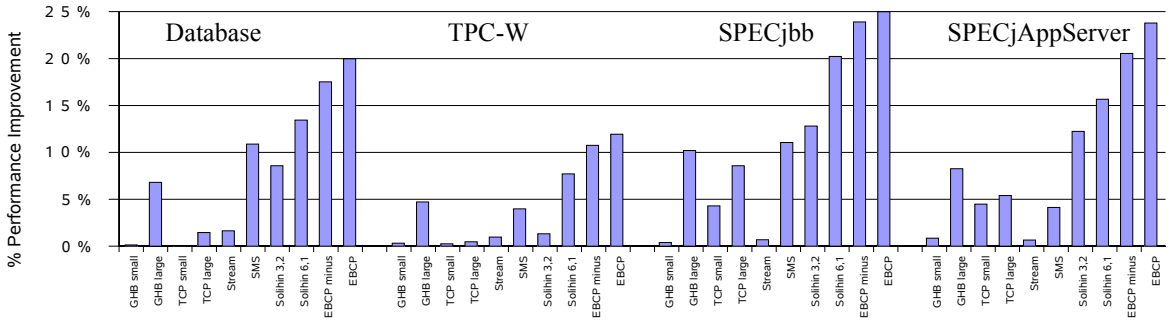

Figure 9: Performance comparison of epoch-based correlation prefetcher (EBCP) with other prefetchers.

The results show that the epoch-based correlation prefetcher significantly outperforms the other prefetching schemes. Among the other prefetching schemes, only *GHB large*, *SMS*, *Solihin 3,2* and *Solihin 6,1* work effectively across the four benchmarks. *TCP large* works fairly effectively for SPECjbb2005 and SPECjAppServer2004 but not for the database workload or TPC-W. Other than *SMS*, none of the schemes that require less than a megabyte of storage to implement, such as *GHB small*, *TCP small* and the stream prefetcher, work effectively on these commercial applications. Their large instruction and data working sets overwhelm the small correlation tables of *GHB small* and *TCP small*, while their irregular data access patterns render the stream prefetcher ineffective. By comparison, *SMS* is area-efficient because the prefetch addresses for each 2KB memory region can be encoded using only four bytes. *SMS* performs relatively well on the database workload and SPECjbb2005 but not on TPC-W and SPECjAppServer2004. It doesn't perform well on the those two benchmarks because it does not prefetch for instruction misses.

Comparing *GHB large* with *Solihin 6,1*, *GHB large* does not perform as well even though they both employ a prefetch depth of six. This is because *Solihin 6,1*'s correlation table is 16 times the size of *GHB* large's correlation table, made possible because its table is stored in main memory instead of in an on-chip structure as in *GHB*. Comparing *Solihin 3,2* with *Solihin 6,1*, the latter achieves superior performance on all four benchmarks, confirming Wenisch et al's observations [16] that depth prefetching is more important than width prefetching, at least for these commercial workloads.

Comparing the epoch-based correlation prefetcher *EBCP* with *Solihin 6,1*, the epoch-based correlation prefetcher outperforms Solihin's scheme on all four benchmarks. On the database OLTP workload, the epoch-based correlation prefetcher achieves an overall performance gain of 20% versus 13% for *Solihin 6,1*. For TPC-W, the gain is 12% for the epoch-based correlation prefetcher versus 8% for *Solihin 6,1*. For SPECjbb2005, the gain is 28% for the epoch-based correlation prefetcher versus 20% for *Solihin 6,1*, while on SPECjAppServer2004, the gain is 24% for the epoch-based correlation prefetcher versus 16% for *Solihin 6,1*. The epoch-based correlation prefetcher performs better than *Solihin 3,2* and *Solihin 6,1* even though their correlation table entries all store the same number of prefetch addresses. This is because the epoch-base correlation prefetcher selects more useful prefetch addresses to store in its correlation table entries. First, it does not store the prefetch addresses of the misses that belong to the same epoch as the triggering miss. Second, it does not store the prefetch addresses of the misses in the next epoch after the triggering miss as the prefetches in this epoch will not be timely (since the epoch of the triggering miss is needed to read the correlation table from main memory and the next epoch is needed for the prefetch requests to issue and return their data). Third, the epoch-based correlation prefetcher's LRU replacement policy for the prefetch addresses in its correlation table entry enables it to dynamically select between prefetch depth and prefetch width based on runtime information for that entry. To demonstrate the benefit of not storing the prefetch addresses of the misses in the next epoch after the triggering miss, the performance of a handicapped version of the epoch-based correlation prefetcher, termed *EBCP minus*, is also shown in Figure 9. This handicapped version does store the prefetch addresses of the misses that belong to the next epoch after the triggering miss. The results shows that *EBCP* consistently outperforms *EBCP minus* across all four benchmarks.

6 Conclusions

This paper demonstrates how insights from the epoch MLP model lead to the design of a highly effective correlation prefetcher whose large correlation table is cost-effectively stored in main memory and which targets the removal of entire epochs instead of individual off-chip misses. The correlation table is accessed in a timely manner by exploiting memory-level parallelism to hide the latency of its access under a previous epoch. This epoch-based correlation prefetcher does not require any on-chip correlation table storage. Its only on-chip structures are the EMAB, the small prefetch buffer and the simple prefetcher control, all of which are off the critical path and require minimal chip area to implement.

Experimental results using four important commercial benchmarks demonstrate that, despite its very low hardware cost, the epoch-based correlation prefetcher significantly outperforms other recently proposed prefetching schemes. It performs well even on traditionally challenging workloads such as on-line transaction processing. The results also show that, except for Spatial Memory Streaming, none of the other prefetchers that require less than a megabyte of on-chip storage work effectively on these commercial benchmarks. In contrast, the epoch-based correlation prefetcher achieves its performance gains by taking advantage of unused main memory to store the large correlation table required by these commercial applications and by taking advantage of un-utilized memory bandwidth to access and update the table. It requires almost no on-chip storage to implement, requires little or no modification to the memory controller, and it can take advantage of industry-standard memory technology such as FBDIMMs and DDR2/3. The epoch-based correlation prefetcher achieves superior performance gains because it is better at removing entire epochs, and the removal of epochs directly translates into improved overall performance.

Due to the on-chip location of its prefetcher control in front of the core-to-L2 crossbar, the epoch-based correlation prefetcher is also amendable to implementation in chip multiprocessors. Future work includes design and evaluation of an epoch-based correlation prefetcher that is optimized for a chip multiprocessor.

References

[1] I. Park, C. Ooi, and V. Vijaykumar, "Reducing Design Complexity of the Load/Store Queue," Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 411, 2003.

[2] L.Barroso, K.Gharachorloo and E. Bugnion, "Memory System Characterization of Commercial Workloads," in Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 3-14, 1998.

[3] J. Lo et. al., "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors," in Proceedings of 25th

Annual International Symposium on Computer Architecture, pp. 39-50, 1998.

[4] K. Keeton et. al., "Performance Characterization of Quad Pentium Pro SMP using OLTP Workloads," in University of California at Berkeley Computer Science Division Technical Report UCB/CSD-98-1001.

[5] R. Hankins et. al., "Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice," in Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 151, 2003.

[6] A. Smith, "Sequential Program Prefetching in Memory Hierarchies," in IEEE Computer, vol. 11, pp. 7-21, 1978.

[7] W. Hsu and J. Smith, "A Performance Study of Instruction Cache Prefetching Methods," in IEEE Transactions on Computers, vol. 47, pp. 497-508, May 1998.

[8] E. Davidson, M. Annavaram and J. Patel, "Call Graph Prefetching for Database Applications," in ACM Transactions on Computer Systems, pp. 412-444, November 2003.

[9] B. Calder, G. Reinman and T. Austin, "Fetch Directed Instruction Prefetching," in Proceedings of 32nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 16-27, 1999.

[10] L. Spracklen, Y. Chou and S. Abraham, "Effective Instruction Prefetching in Chip Multiprocessors for Modern Commercial Applications," in Proceedings of 11th International Symposium on High-Performance Computer Architecture, pp. 225-236, 2005.

[11] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," in Proceedings of 17th Annual International Symposium on Computer Architecture, 1990.

[12] S. Palarcharla and R. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", in Proceedings of 21st Annual International Symposium on Computer Architecture," pp. 24-33, 1994.

[13] J. Fu et al, "Stride Directed Prefetching in Scalar Processors", in Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 102-110, 1992.

[14] T. Chen et al, "Effective Hardware-Based Data Prefetching for High Performance Processors," in IEEE Transactions on Computers, pp. 609-623, 1995.

[15] Z. Hu et al, "Tag Correlating Prefetchers," in Proceedings of 9th International Symposium on High-Performance Computer Architecture, pp. 317-327, 2003.

[16] T. Wenisch et al, "Temporal Streaming of Shared Memory," in Proceedings of 32nd Annual International Symposium on Computer Architecture, 2005.

[17] J. Baer, "Dynamic Improvements of Locality in Virtual Memory Systems," in IEEE Transactions on Software Engineering, March 1976.

[18] J. Pomerene et al, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," U.S. Patent 4,807,110, February 1989.

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

[20] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," in Proceedings of 24th Annual International Symposium on Computer Architecture, pp. 252-263, 1997.

[21] T. Sherwood, S. Sair and B. Calder, "Predictor-Directed Stream Buffers," in Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 42-53, 2000.

[22] T. Alexander and G. Kedem, "Distributed Predictive Cache Design for High Performance Memory Systems," in Proceedings of 2nd International Symposium on High-Performance Computer Architecture, pp. 254-263, 1996.

[23] A. Lai, C. Fide and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," in Proceedings of 28th Annual International Symposium on Computer Architecture, pp. 144-154, 2001.

[24] Y. Solihin, J. Lee and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," in Proceedings of 29th Annual International Symposium on Computer Architecture, 2002.

[25] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," in Proceedings of 10th International Symposium on High-Performance Computer Architecture, 2004.

[26] Y. Chou, B. Fahs and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in Proceedings of 31st Annual International Symposium on Computer Architecture, 2004.

[27] Y. Chou, L. Spracklen and S. Abraham, "Store Memory-Level Parallelism Optimizations for Commercial Applications," in Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, 2005.

[28] S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches Using Spatial Footprints," in Proceedings of 25th Annual International Symposium on Computer Architecture, 1998.

[29] A. Glew, "MLP yes! ILP no!," in ASPLOS Wild and Crazy Idea Session '98, October 1998.

[30] "IBM Unleashes World's Fastest Chip in Powerful New Computer", at http://www-03.ibm.com/press/us/en/pressrelease/21580.wss.

[31] D. Sorin et. al., "Analytic Evaluation of Shared-Memory Systems with ILP Processors", in Proceedings of 25th Annual International Symposium on Computer Architecture", 1998.

[32] http://www.micron.com/products/modules/ddr2sdram/fbdimm.html

[33] Sun Microsystems. SPARC Architecture Manual V9, 1996.

[34] www.tpc.org

[35] www.spec.org

[36] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos, "Spatial Memory Streaming," in Proceedings of 31st Annual International Symposium on Computer Architecture, 2006.

[37] D. Perez, G. Mouchard, O. Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms," in Proceedings of 37th International Symposium on Microarchitecture, 2004.

[38] H. Modi, L. Spracklen, Y. Chou, S. Abraham, "Accurate Modeling of Aggressive Speculation in Modern Microprocessor Architectures," in Proceedings of MASCOTS 2005, 2005.

[39] W. Lin, S. Reinhardt and D. Burger, "Reducing DRAM Latencies With An Integrated Memory Hierarchy Design," in Proceedings of 7th International Symposium on High Performance Computer Architecture, 2001.

[40] Z. Wang et. al., "Guided Region Prefetching: A Cooperative Hardware/Software Approach," in Proceedings of 30th Annual International Symposium on Computer Architecture, 2003.

[41] M. Ferdman and B. Falsafi, "Last-Touch Correlated Data Streaming," in Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2007.