Stream Chaining: Exploiting Multiple Levels of Correlation in Data Prefetching*

Pedro Díaz and Marcelo Cintra
School of Informatics
University of Edinburgh
pedro.diaz@ed.ac.uk, mc@staffmail.ed.ac.uk

ABSTRACT

Data prefetching has long been an important technique to amortize the effects of the memory wall, and is likely to remain so in the current era of multi-core systems. Most prefetchers operate by identifying patterns and correlations in the miss address stream. Separating streams according to the memory access instruction that generates the misses is an effective way of filtering out spurious addresses from predictable streams. On the other hand, by localizing streams based on the memory access instructions, such prefetchers both lose the complete time sequence information of misses and can only issue prefetches for a single memory access instruction at a time.

This paper proposes a novel class of prefetchers based on the idea of linking various localized streams into predictable chains of missing memory access instructions such that the prefetcher can issue prefetches along multiple streams. In this way the prefetcher is not limited to prefetching deeply for a single missing memory access instruction but can instead adaptively prefetch for other memory access instructions closer in time.

Experimental results show that the proposed prefetcher consistently achieves better performance than a state-of-the-art prefetcher – 10% on average, being only outperformed in very few cases and then by only 2%, and outperforming that prefetcher by as much as 55% – while consuming the same amount of memory bandwidth.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Design Styles

General Terms

Performance

Keywords

Data Prefetching

1. INTRODUCTION

A well known performance bottleneck is the so-called memory gap (or wall), which refers to the speed difference between microprocessor execution and memory access latency, typically in the order of a few hundred processor cycles nowadays. In the past, semiconductor and microarchitectural advances allowed microprocessors to increase operating frequency at a steady rate, while the most significant advances in memory technology had been in density, not speed. While recent semiconductor scaling trends have greatly reduced the previous acceleration pattern of the memory gap, the gap as seen by microprocessors is still likely to continue increasing. In fact, fast increasing microprocessor operating frequencies have now been exchanged for fast increasing number of microprocessor cores per chip. This has had a similar adverse effect on the performance of the memory subsystem. On the one hand, wire latencies and power constraints limit significant increases in the size of on-chip caches, such that these are expected to increase sub-linearly with the number of cores. On the other hand, packaging limitations and signal delays to off-chip also lead to an expected sub-linear increase in memory bandwidth with respect to the number of cores. Both of these effects combined put increased pressure on the memory subsystem and increase the average off-chip access times as observed by the cores.

Hardware prefetching has long been a successful technique to help overcome the memory gap. Current hardware prefetching algorithms attempt to predict future processor memory accesses by keeping a history of recent cache misses. Prefetchers have to deal with several challenges, which include: 1) entropy of information in the miss stream leading to limited predictability; 2) irregularity in the miss stream due to order changes in the misses and spurious, infrequent, misses leading to further reduction in predictability; 3) variation in the time distance between misses leading to timeliness issues: prefetches may arrive too late, thus not completely hiding the miss latency, or too early, thus replacing other useful data from the cache or prefetch buffer; and 4) physical limitations of hardware implementations exacerbating the problems above. A common way to cope with these challenges is to divide the global miss address stream into multiple streams – a process we call *localization* [1]. With localization, misses are grouped according to some property and the expectation is that the resulting streams are more predictable than the original global stream. Once the localized streams have been generated, various mechanisms for detecting patterns in them can be used, such as strides, deltas, Markov chains, and even straightforward repetition of a sequence of addresses; the choices of the localization approach and the pattern correlation mechanism are mostly orthogonal. For instance, common stride prefetchers [2, 4] and the predictor in [12] separate the stream of misses according to the PC of the missing memory access instruction, which we call *PC localization* and is an instance of *execution context localization*. The predictors in [13] and [19]

*This work was supported in part by the EC under grants IP 27648 (FP6) and HiPEAC IST-004408.

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

[1]We use the term *localization* to refer to the act of assigning misses to a particular stream in order to distinguish it from the *correlation* properties that may exist in the stream(s).

separate the stream of misses according to memory address ranges of the misses, which we call *spatial localization*. The predictor in [22] groups misses according to their appearing together within a time period, which we call *temporal localization*.

Ideally, the choice of property used for localization exploits some inherent property of programs and data access patterns that lend the resulting streams more predictable. For instance, PC localization exploits the fact that different static memory access instructions perform different functions from their neighbor instructions in both execution time and instruction address space: e.g., the load of a "`p->next`" type of operation in a list traversal and the loads/stores used in the neighboring computation. Similarly, spatial localization exploits the fact that regular datastructures are commonly laid out in a continuous portion of the address space and are usually accessed in some regular pattern, possibly different from the pattern used to access data that is used at around the same time but belonging to other datastructures. Unfortunately, localization creates two problems: i) most forms of localization break the chronological order of misses, possibly leading to timeliness problems and ii) the resulting streams may be too short, possibly leading to uncovered misses across stream boundaries and also to timeliness problems. For instance, the misses from a PC localized stream may be interleaved with many misses from other PC localized streams so that prefetches are issued too early in time displacing useful data from the cache. Similarly, the misses from a space localized stream may be interleaved with some misses to addresses outside its address range. Short streams make it difficult to exploit deep and wide correlation patterns such as Markov chains.

In this paper we propose the idea of chaining localized streams in order to both *partially* reconstruct the chronological information in the miss stream and to build longer streams, while maintaining a higher miss prediction accuracy derived from localization. In particular, we apply this idea – which we call *Stream Chaining* – to a PC localized prefetcher and we show one scheme for chaining streams, which we call *Miss Graph prefetching*. We show that this scheme captures well common application behavior and we show that it can be easily implemented on a simple extension to the Global History Buffer (GHB) [12], which is a convenient structure to implement various prefetching algorithms. The paper focuses on the lowest level on-chip data cache, as this is the last level before hitting the memory wall. We evaluate the proposed prefetcher with benchmarks from the SPEC 2006 and BioBench suites and compare it against both a state-of-the-art PC based localization prefetcher – PC/DC [12] – and its non-localized counterpart – G/DC [9, 12]. Experimental results show that the proposed prefetcher consistently achieves better performance than PC/DC – it is only outperformed in very few cases and then by only 2%, and it outperforms PC/DC by as much as 55% while consuming the same amount of memory bandwidth. This is due to much improved accuracy and timeliness of the prefetches issued.

The rest of the paper is organized as follows. Section 2 discusses prefetching algorithms, and in particular localization, and introduces the GHB data structure. Section 3 introduces the concept of stream chaining and presents the miss graph prefecher which is based on it. Section 4 describes the evaluation methodology and Section 5 presents experimental results. Related work is discussed in Section 6 and conclusions appear in Section 7.

2. OVERVIEW OF PREFETCHING MECHANISMS

2.1 Miss Localization and Correlation

A common strategy for building a data prefetching algorithm is to keep a history of past memory accesses. Prefetchers then use a correlation algorithm to predict the next memory reference based on this history. However, having a single global history of past memory accesses is not usually effective: program complexity, different execution phases, and random non-predictable memory accesses degrade the quality of the prefetching predictions. To overcome these problems, current state-of-the art prefetchers use a technique known as miss stream localization.

With miss stream localization past memory accesses (cache misses) are classified into groups according to a pre-set criterion. These groups, known as address streams, are then used to make predictions about future memory accesses. Popular options for miss stream localization include grouping misses according to the PC of the instruction that generated them [2, 4, 12], or to the region of memory they reference [13, 14, 19], or to some period of time in which the misses occur [22], or even to previously executed branch instructions [21]. Figure 1 depicts a simple example of a global miss address stream and how it can be localized following PC, spatial, and temporal localization. With memory access instruction PC localization, upon a miss caused by a certain memory access instruction only previous miss addresses generated by the same instruction are used to predict the address(es) to prefetch. This localization scheme thus attempts to uncover and exploit some PC correlation present in the application memory access pattern. Thus, as depicted in the figure, the stream of PC_A will lock onto its access pattern, regardless of where in the address space its data is located and regardless of what other misses appear concomitantly with its misses. However, the PC localized prefetch predictor in this figure will not realize that a miss to $A2$ tends to happen after a miss to $A1$. With spatial localization, upon a miss to a data address within some address range region only previous miss addresses generated to the same range are used to predict the address(es) to prefetch. This localization scheme thus attempts to uncover and exploit some spatial correlation present in the application memory access pattern. Thus, as depicted in Figure 1, the stream of misses to the address region at the bottom of the address space – $A11$ to $A14$ – will lock onto its sequential access pattern, regardless of what memory access instructions are used to perform the accesses and regardless of what other misses appear concomitantly with its misses. However, the spatial localized prefetch predictor in this figure will not realize that a miss to $A1$ tends to happen after $A12$ and may not even realize that an access to $A13$ does not really occur in practice. Finally, with temporal localization, upon a miss only previous miss addresses generated at around the same time as the previous occurrence of this miss are used to predict the address(es) to prefetch. This localization scheme thus attempts to uncover and exploit some temporal correlation present in the application memory access pattern. Thus, as depicted in the figure, the stream of misses to $A4$ and $A6$ may be detected regardless of what memory access instructions are used to access them and regardless of where they are physically located in the data address space. However, the temporal localized prefetch predictor in this figure may not realize that misses to both $A7$ and $A11$ tend to follow shortly after misses to $A1$, even though there might be some alternating pattern to it. While the example in this figure is simple and somewhat artificial, it demonstrates both that there is less entropy in the localized patterns than in the global miss stream and that different localization approaches excel and struggle at different access and data layout patterns.

Once the address stream has been localized somehow, prefetchers use correlation algorithms to try to find patterns in past memory accesses. A simple correlation algorithm, known as address correlation, searches the history looking for other occurrences of the latest miss or group of misses. If such repetitions (correlations) are found, then the algorithm prefetches the next address recorded after the correlation. A more refined correlation algorithm known as delta correlation calculates differences (deltas) between the latest

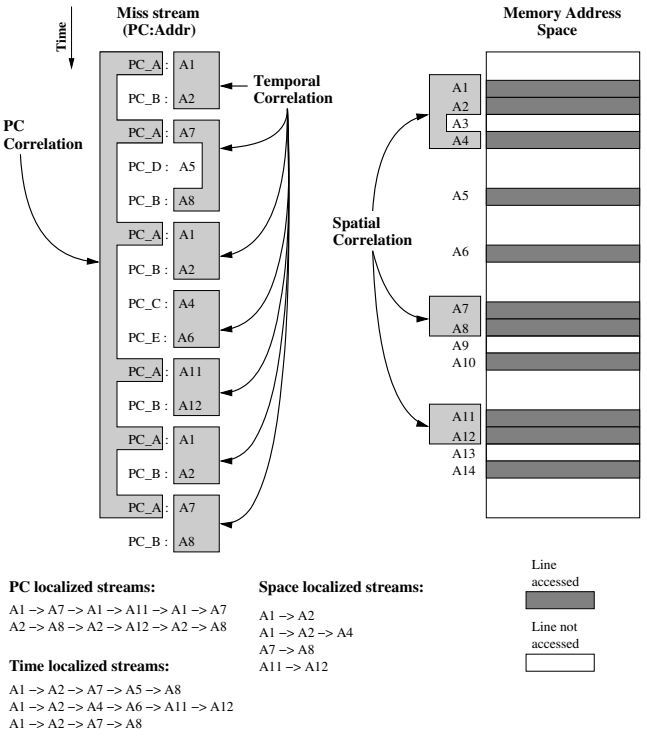

Figure 1: Examples of PC, spatial, and temporal localization approaches.

miss addresses and searches for similar differences in the past miss history [9].

Finally, once a correlation has been made, one or more prefetch requests are made for the particular miss event. The number of prefetch requests issued per miss event is known as the *prefetch degree*. The greater the prefetch degree the more future misses that will be possible to avoid. On the other hand, if the prefetch degree is too large issues such as bandwidth saturation, cache pollution, and increased inaccuracies commonly degrade performance.

Based on the mostly orthogonal stages of localization and correlation, a taxonomy to classify prefetching algorithms was proposed in [12]. In this taxonomy, each algorithm is denoted as a pair `X/Y`, where `X` denotes the method used to localize the miss address stream and `Y` is the algorithm used to correlate memory accesses within a stream. For instance, a simple stride prefetcher with `PC` based localization and `constant stride` correlation would be referred to as `PC/CS`, and the best prefetcher in [12] uses `PC` based localization and address `delta correlation` and is then referred to as `PC/DC`. A prefetcher that does not localize the miss address stream in any way is referred to as `global`, so that for instance the Markov predictor of [8] is referred to as `G/AC` (`AC` stands for address correlation).

2.2 The Problem of Accuracy and Timeliness in Localized Prefetchers

From the discussion in the previous section it is clear that each localization method has advantages and disadvantages. Three criteria are normally used to characterize the performance of a prefetching algorithm: *coverage*, *accuracy*, and *timeliness*. Coverage is defined as the ratio of issued prefetches to the total number of misses produced in a non-prefetching environment. Accuracy measures the ratio of useful (i.e., eventually used by the program) prefetches to the total number of prefetches issued. The third criterion, timeliness, is more difficult to measure. A prefetch request is normally considered to be timely if it brings in the data in time before it is needed. However, a more comprehensive classification of timeliness also takes into account whether data is prefetched too early, as this may create cache pollution by evicting other data that is still needed.

Schemes based on localizing streams according to the PC of memory access instructions that generate the misses can potentially lead to higher performance gains than those that treat all misses as a single global stream [12, 15], although this varies according to individual applications. Their key advantage is that they can take into account the execution context and focus their correlation algorithms on streams of predictable misses from individual memory access instructions. They can then filter out less predictable misses and spurious misses that appear in the global stream due to variable interleavings of misses.

It is important to note that entries in the same PC localized miss stream are ordered chronologically, but two chronologically consecutive misses will not necessarily belong to the same miss stream. Thus, if a prefetching algorithm issues a number of prefetching requests based on a PC localized stream, each prefetched data item for a dynamic instance $\texttt{PC}_i^n$ is expected to be used in following instances $\texttt{PC}_i^{n+1}$ to $\texttt{PC}_i^{n+d}$, where d is the degree of prefetching, and before memory accesses from some other $\texttt{PC}_j$ [2]. However, an indeterminate number of memory accesses can appear between the prefetch trigger at $\texttt{PC}_i^n$ and the use of the prefetched data by $\texttt{PC}_i^{n+1}$ to $\texttt{PC}_i^{n+d}$, as long as these requests are not classified in the same localized stream of $\texttt{PC}_i$.

Naturally, the problem mentioned above is worse with larger prefetch degrees. Unfortunately, to increase coverage and amortize long memory access times, these prefetchers often resort to large degrees of prefetching [12]. The final result is that this leads to two unfortunate behaviors. First, prefetching a large number of lines for the same static memory access instruction may lead to some of these being incorrectly prefetched, either because the pattern changes or because not as many lines are actually required. This, in turn, leads to reduced accuracy for the deeper prefetch requests. Second, by expecting to hide long memory access latencies the prefetcher may err by being too aggressive and prefetching too far in advance. Such prefetched data brought to the cache in an untimely (i.e., premature) manner can evict data that is needed before it. Additionally, prefetched data brought too early to the cache may itself be evicted before it is used. Both problems result in a waste of memory bandwidth which may result in application slowdown compared to non-prefetching setups.

2.3 The Global History Buffer

Many (hardware) data structures can be used to implement different prefetching algorithms, but [12] has proposed a common data structure that is flexible enough to allow efficient implementation of a number of prefetching algorithms. This structure, called the Global History Buffer (GHB), is a FIFO-like structure that stores past cache miss addresses in chronological order. Each entry in the GHB also contains a pointer that allows GHB entries to be connected in time-ordered linked lists forming the different localized miss streams. A global Head Pointer points to the head of the GHB. Additionally, an Index Table (IT) is used to access the most recent address for each stream. The IT is indexed using a key appropriate for the localization scheme used (e.g., the PC of the memory access instruction that generated the miss), which allows for the implementation of different localization schemes. Figure 2a shows an example GHB for the PC/DC prefetcher [12]. A detailed description of a hardware implementation of delta correlation using the GHB can be found in [13]. In Section 3.2 we show how the

[2] In our representation we use subscripts to refer to different static instructions and superscripts to refer to dynamic instances of static instructions.

same structure can be used to implement our proposed prefetcher. The GHB offers three important advantages over other methods to store miss history, such as tables: reduction of stale data, a more complete history, and lower storage requirements [12].

3. STREAM CHAINING

In this section we present the *stream chaining* approach to data prefetching, which adds a third level of design choice to prefetching schemes. We start by presenting the general idea in Section 3.1. In Section 3.2 we present one specific prefetcher that uses the stream chaining approach, namely the *Miss Graph prefetcher*. To accommodate the new level of operation, we extend the taxonomy of [12] with a third term, so that prefetching schemes are denoted by a triple `X/Y/Z`, where `X` and `Y` are as before, and the new term `Z` denotes the method used to link streams into groups. We use `MG` to refer to our proposed prefetcher of Section 3.2.

3.1 Overview

As already mentioned, the main problem with PC localized prefetching schemes is that there is no chronological information relating the streams (the same is true of other localization schemes, such as the address region localization of [13]). This can lead to prefetches sometimes being triggered along streams whose memory accesses appear too far apart in time, leading to untimely (premature) prefetches. While the GHB does contain the total timing relation among individual memory accesses of all streams, this information is not in a format that can be readily used. The key to reconstructing appropriate timing information across streams is to provide a suitable stream chaining algorithm.

Our proposal is to link streams in a way that *partially* reconstructs the chronological information in the miss stream, such that the result corresponds directly to the *common path of misses* followed by the application. Note that in this way, what is reconstructed are sequences of *streams*, which are different from the complete sequence of *individual misses* that is found in the complete global miss stream. Another way of thinking about stream chaining is that it attempts to predict what miss stream will be activated next in program order. In this way, a three-level prefetcher with stream chaining can predict not only the expected next misses from the current memory access instruction but also the expected next misses from the expected next memory access instructions to miss in program order. Thus, such a prefetcher has an extra level of flexibility and can adapt to situations both in which missing memory accesses from the same static instruction are too far apart and in which missing accesses from static instructions consecutive in program order are too near. This additional level of adaptiveness can potentially improve both timeliness and accuracy with respect to traditional deep two-level prefetching.

Based on our empirical results, the flow of missing memory access instructions commonly follows stable and repeatable patterns. These patterns can be represented by a directed graph where nodes correspond to static memory access instructions (i.e., streams) and edges correspond to links between two streams, indicating that a miss in one stream is likely to be followed by a miss in the other stream. Figure 2b shows the localized streams and one possible chaining of them for the complete miss stream and GHB state in Figure 2a. While simple in nature, generating graphs that represent the core flow of missing memory access instructions and excludes spurious misses from memory access instructions, either in infrequent control paths or that generate only occasional misses, is not trivial and is the key to a good stream chaining prefetcher. In this example linking the streams of $\texttt{PC}_A$ and $\texttt{PC}_B$ could be deemed inappropriate by the algorithm, as it corresponds to an infrequent flow of misses (Figure 2 does not show explicitly a stream for $\texttt{PC}_B$ to keep the figure small, but it is assumed that this instruction does execute and cause misses occasionally). Note that the resulting graph does not then contain all the possible links from the total miss sequence information in the global miss stream, but only a carefully selected set of those. Constructing the complete graph is both impractical from the point of view of storage and processing time and yields too much information that can distract the algorithm from finding the most likely miss patterns.

The fact that the memory access chaining algorithm cannot and should not keep all links results in the graphs being disconnected or some graphs being acyclic. Note that in this way, using the information about linked streams to predict the next missing memory access instruction along the link may or may not be the same as predicting the next miss. In this example, the given chaining of streams allows a prefetcher on a miss from $\texttt{PC}_A$ to prefetch not only the next values to be consumed by $\texttt{PC}_A$ itself but also the next value(s) to be consumed by $\texttt{PC}_D$ (the next instruction likely to miss) or even $\texttt{PC}_E$. Alternatively, since $\texttt{PC}_A$ is in a cyclic graph, if the distance between its consecutive instances is too large then it could simply rely on a peer (such as $\texttt{PC}_D$ and $\texttt{PC}_E$) to prefetch the data it will need next.

With the addition of stream chaining to a baseline PC/DC, a prefetcher can exploit three types of correlation in the miss stream: PC correlation (from the localization), spatial correlation (from the delta correlation), and temporal (from the stream chaining).

3.2 Miss Graph Prefetching

In this section we present one prefetcher that uses the stream chaining approach. Every stream chaining prefetcher has to deal with two implementation issues: how to link streams (Section 3.2.1) and how to use the graph to issue prefetches (Section 3.2.2).

3.2.1 Graph Construction

As mentioned in the previous section, the key to effective stream chaining is the choice of which links to use and which to ignore. In fact, with a large enough GHB the complete chronological reconstruction of the miss stream would result in an unmanageable graph. In this section we present a simple scheme to stream chaining that results in relatively small graphs that capture the majority of miss sequences in steady-state execution (see Section 5.3). The scheme can also be easily implemented using the GHB structure. Naturally, this is only a first attempt at a novel class of prefetching schemes and other stream chaining mechanisms are possible, including Markov chains.

In GHB-based prefetchers the Index Table (IT) holds pointers to every localized stream in the GHB (Section 2.3). Thus, chaining streams corresponds to linking entries in the IT. To do this, we extend the IT by adding a new pointer to each IT entry – `NextIT` – which points to the IT entry corresponding to the stream that is expected to be activated next. The `NextIT` field also includes an additional bit to signal if this pointer is valid. To identify strong (i.e., stable) links, we also add a saturating counter to each IT entry – `Ctr`. The operation of this counter is explained next. Finally, we also add a new global register to the IT – `PreviousIT` – which is also a pointer to an IT entry. Figure 2a shows the GHB extensions in grey.

Our miss stream chaining algorithm builds a graph of past (temporal) correlations between localized streams as follows. Initially, `NextIT` is invalid and `Ctr` is set to zero on all entries of the IT. As misses occur, the IT and the GHB are populated as described in Section 2.3 and [12]. The new `PreviousIT` pointer is left pointing to the last stream to suffer a miss (i.e., last IT entry used). Then for a subsequent miss that activates the IT entry `IT[cur]`, we check the previous IT entry using `IT[PreviousIT]` and perform the following operations:

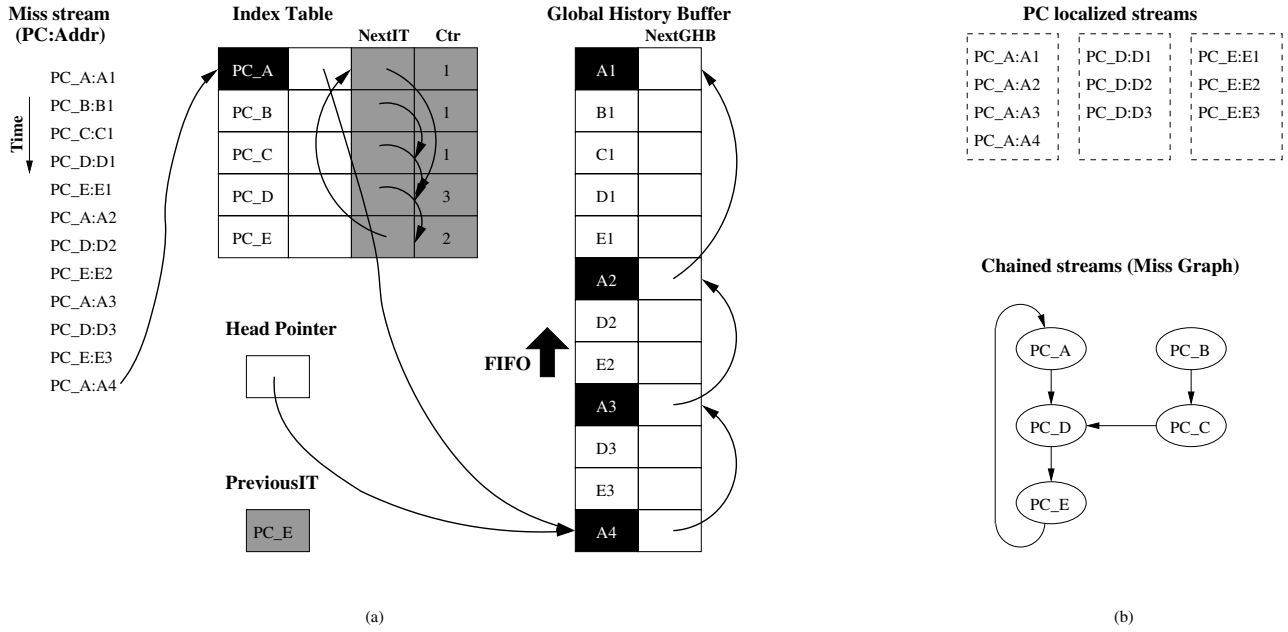

Figure 2: PC/DC and PC/DC/MG example: (a) GHB state for both prefetchers (entries shown in grey are extensions to the design of [12] and are described in Section 3.2); (b) streams localized according to the missing memory access instruction's PC (streams for PC_B and PC_C are not shown for simplicity) and possible chaining of the streams.

```
if IT[PreviousIT]→NextIT is invalid then
    IT[PreviousIT]→NextIT = IT[cur]
    IT[PreviousIT]→Ctr = 1
else
    if IT[PreviousIT]→NextIT == IT[cur] then
        IT[PreviousIT]→Ctr++ (saturating increment)
    else
        IT[PreviousIT]→Ctr−−
    end if
    if IT[PreviousIT]→Ctr == 0 then
        IT[PreviousIT]→NextIT = IT[cur]
        IT[PreviousIT]→Ctr = 1
    end if
end if
PreviousIT = cur
```

By following these operations, the `NextIT` pointers in the IT form a directed graph, which can be cyclic or acyclic, can be disconnected, and in which there is only one outgoing edge from each node but possibly more than one incoming edge to a node. Figure 2a shows the state of the `NextIT` pointers, the `Ctr` counters, and the `PreviousIT` pointer just before the miss to $A4$ is processed. The corresponding graph is shown in Figure 2b.

The graph constructed with the mechanism described shows a history of correlations between localized miss streams, showing which IT entry followed which in the past. The role of the saturating counters `Ctr` is to provide hysteresis and protection from noise from sporadic misses: by setting a minimum threshold to `Ctr` we obtain a graph with only the most stable transitions between localized streams. Next we explain in some more detail how a prefetcher can use the extended GHB entries to prefetch along nodes in the graph.

3.2.2 Prefetch Operation

With the extended entries in the IT representing the miss graph, our proposed prefetcher operates as follows. First, the prefetcher identifies the current miss stream, which simply involves searching the IT for an entry that matches the PC of the current missing memory access instruction. Here, unlike PC/DC, which would follow the IT pointer into the corresponding GHB stream, our prefetcher identifies the next stream to prefetch for by following the `NextIT` pointer in the current IT entry. So, for instance, a miss from a memory access instruction at $\texttt{PC}_A$ in Figure 2 will first follow the corresponding `NextIT` pointer to the stream of $\texttt{PC}_D$. For every stream that the prefetcher attempts to prefetch for, it follows the IT pointer into the corresponding GHB stream and then follows the links in the GHB entries to attempt to establish a correlation among the miss addresses. In our PC/DC/MG prefetcher we use *delta correlation* on the addresses. If a correlation is found along the stream the prefetcher issues one prefetch along this stream. Thus, for each stream, PC/DC/MG behaves as a PC/DC with a prefetch degree of one. After issuing a prefetch for a given stream, the prefetcher again follows the `NextIT` pointer to the next stream to prefetch for and repeats the steps for prefetching for this stream. This process is repeated for a number of times equal to the prefetching degree parameter. In order to avoid following "weak" links into new streams, we impose a minimum threshold on the `Ctr` value below which the prefetcher will not follow the `NextIT` and will stop prefetching from further streams.

The graph construction we propose in Section 3.2.1 leads to graphs where the outgoing edge degree is no greater than one and graphs can be cyclic. Thus, starting from some node, the chains with *degree* nodes created by following the operations just described are either a linear sequence of distinct nodes or a cycle. Further, the linear sequences can either have a number of distinct nodes greater than or equal to the degree of prefetching plus one, or a number of distinct nodes smaller than the degree plus one (the "plus one" comes from the fact that we skip the initial node). Figure 3 shows the three possible cases of graphs.

For graphs as in Figure 3a the operation of the prefetcher as described above is complete. For the other two cases of graphs the operation must be slightly extended. For graphs as in Figure 3b if we want to issue as many prefetches as the degree allows us, we would have to follow an edge back to some streams for which we have already issued a prefetch. The problem in this case is that the prefetcher would have to remember, for every revisited stream, the correlation used the last time around and the resulting address prefetched, in order to find the next delta and to compute the next address to prefetch. Instead, a simpler solution is to add a pre-

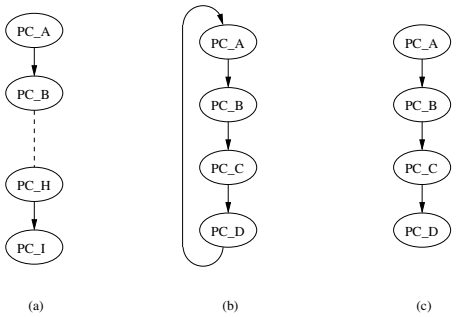

Figure 3: Miss graph cases: (a) non-cyclic chain longer than the prefetch degree; (b) cyclic chain shorter than the prefetch degree; and (c) non-cyclic chain shorter than the prefetch degree. The prefetch degree is assumed to be 8 and the current miss is generated by $\texttt{PC}_A$.

pass stage where we quickly follow the "strong" `NextIT` pointers to identify whether the graph is cyclic or a long enough sequential chain. Then, if the graph is cyclic we perform the steps above except that we compute the correlations and issue more than one prefetch per stream, where each stream gets an equal share of the prefetch degree (or nearly equal when the number of nodes in the chain does not divide the prefetch degree). For graphs as in Figure 3c we can simply convert them to the case of Figure 3b by pretending that there is a back edge from the end of the chain to its beginning. Finally, if the graph consists of only the entry node then instead of starting from the next stream (which is unavailable) we simply prefetch for the current stream, basically reverting to PC/DC behavior.

Since the prefetches for the different streams are generated in a single prefetcher activation, one optimization that is possible in the cases of Figures 3b and 3c is to issue the prefetches to the memory sub-system such that prefetches to consecutive streams are interleaved. Thus, for instance, in the case of Figure 3b we can order the prefetch requests such that the first prefetch request for $\texttt{PC}_C$ appears right after the first request for $\texttt{PC}_B$ and the second prefetch request for $\texttt{PC}_C$ appears after the second prefetch request for $\texttt{PC}_B$. This ordering is likely to be a better match to the order in which the prefetched data will be needed.

The proposed prefetching mechanism described above is our first attempt at stream chaining based prefetching. One possible advantage of three-level prefetchers with stream chaining is that given a fixed prefetching degree budget one can divide this budget in different ways between *width* – i.e., the number of streams prefetched for – and *depth* – i.e., the number of prefetches along each stream. For instance, if deeper prefetching gives diminishing returns due to too early prefetches or decreasing accuracy, then the prefetcher can issue fewer prefetches from more streams. Alternatively, if the links between streams are too weak then the prefetcher can issue more prefetches from fewer streams. Also, we only classify links as "strong" or "weak", but one could consider finer classifications and adapt the depth for each stream depending on the "strength" of the links followed.

As with other prefetchers [12, 13], in order to avoid prefetched data modifying the natural stream of misses from the program a 1-bit prefetch tag is added to each cache line. This bit is set to one only in lines that come into the cache from a prefetch request and it remains set as long as the line has not yet been used. When such a line is used, a "fake" miss signal is sent to the prefetcher and the bit is reset. The prefetcher then updates its internal data structures as if it were a real miss, but no prefetch request is issued.

3.2.3 Hardware and Operation Complexity

As described in Section 3.2.1, our prefetcher uses an extension to the GHB structure. The additional storage our prefetcher requires are the `NextIT` and `Ctr` for each IT entry and a single `PreviousIT` register. As observed in [12] and in our own experience, both an IT and a GHB with 512 entries each are sufficient to capture the prefetching working set of most applications. In this case, each `NextIT` and the `PreviousIT` consist of 9 bits, as do the other pointers in the original IT and GHB, including the Head Pointer. Our experiments show that small saturation limits for `Ctr` are sufficient to capture stable links between streams and we use 3 bits. Assuming a 32 bit PC the total hardware storage requirements of the original and extended IT/GHB structures are: $512 * (32 + 9) + 512 * (32 + 9) + 9 = 41,993$bits (or approx. 6KBytes) and $512 * (32 + 9 + 9 + 3) + 512 * (32 + 9) + 9 + 9 = 48,146$bits (or approx. 7KBytes). The additional storage requirement of our prefetcher is then less than 1KByte.

One drawback of GHB-based prefetchers is the time it takes to follow links to establish a correlation. With stream chaining, a prefetcher requires following links in multiple streams, which may further increase prefetcher operation time. The increase in operation time in comparison with the single-stream counterpart will depend on the common number of nodes in the miss graph, which in turn depends on the application. Our results suggest that the number of nodes is relatively small (Section 5.3). In case this overhead does become a bottleneck, we note that it is possible to search for correlations in some number of streams in parallel, at the cost of replicated hardware logic. In our experiments, however, we searched for correlations from each stream sequentially.

4. EVALUATION METHODOLOGY

4.1 Prefetching Algorithms

We use PC/DC [12] as a representative of a modern high-performance prefetcher. This prefetcher is based on PC stream localization and was shown to consistently outperform other localized and non-localized prefetchers [12, 15]. We also use a G/DC prefetcher in order to assess the impact of PC based localization. We evaluate our PC/DC/MG prefetcher as described in Section 3.2. With all prefetchers we vary the degree of prefetching from 1 to 16. For PC/DC/MG we use a `Ctr` with 3 bits and set the threshold for "strong" links at 3. All prefetchers first perform a lookup in L2 to check if the lines about to be prefetched are not already there, in which case the prefetch is dropped. In all cases, access to the GHB is exclusive and preemptive, such that a new request forces a previous request currently using the GHB to be dropped.

4.2 Simulation Environment

We conduct our experiments with the SESC simulator [16]. The system we simulate is a 4-issue out-of-order superscalar processor with separate L1 instruction and data caches and a unified L2 cache on chip. All caches are non-blocking. The main microarchitectural parameters are listed in Table 1. The 256KByte L2 cache configuration evaluated is representative of the cache share expected for a processor in a loaded multi-core environment, while the 2MByte L2 cache configuration reflects the case when a single processor is active. The access times for the different L2 cache sizes were computed using CACTI 4.2 [17].

4.3 Benchmarks

We use programs from the SPEC CPU 2006 and BioBench [1] benchmark suites running the Reference data set, except *clustalw* for which we used *hmmer*'s input, which is larger. Some benchmarks were left out because of the non-full-system nature of our simulation environment, which suffered from compatibility problems with libraries, run-time systems, and OS calls required by these applications. The programs were compiled with the GCC 3.4 compiler at the O3 optimization level. We fast-forward simulation

Parameter	Value
Core Frequency	5GHz
Fetch/Issue/Retire Width	6/ 4/ 4
I-Window/ROB	80/ 152
Branch Predictor	64Kbit 2BcgSkew
BTB/RAS	2K entries, 2-way/ 32 entries
Minimum misprediction	20 cycles
Ld/St queue	108
L1 ICache	64KB, 2-way, 64B lines, 2 cycles
L1 DCache	64KB, 4-way, 64B lines, 2 cycles
L1 MSHR's	4
L1-L2 bus	64bits
L2 Cache	256KB/2048KB, 8-way, 64B lines, 13/18 cycles
L2-Memory Bus	64bits, 1.25Ghz
Main Memory	400 cycles
Prefetch degree	1/4/8/16
IT	512 entries, 1 cycle
GHB	512 entries, 5+1*hop cycles

Table 1: Architectural parameters.

for the first 1 billion instructions and we then simulate in detail and collect statistics for the next 1 billion instructions.

5. EXPERIMENTAL RESULTS

5.1 Overall Performance

5.1.1 Cache Size Sensitivity

Figure 4 shows how the relative performance (execution time) changes as the L2 cache size increases in a system *without* prefetching. Performance is relative to that of a system with an ideal L2 cache (i.e., 100% hit rate).

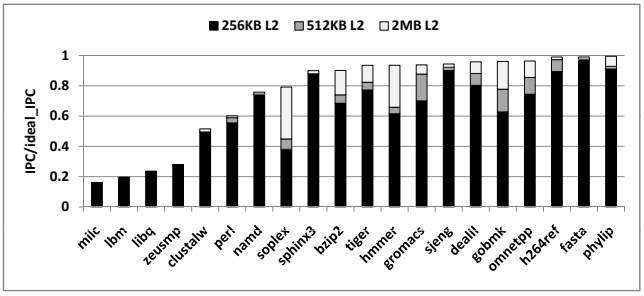

Figure 4: Relative performance versus L2 size *without* prefetching.

From this figure we see that 4 out of the 20 benchmarks achieve performance within 10% of the ideal L2 performance for a 512KB L2 cache. These applications are less likely to benefit as much from prefetching and any such benefits could likely be alternatively achieved simply with a larger L2 cache. On the other hand, many of the other benchmarks fail to come within a reasonable margin of the ideal L2 performance even with a fairly large 2MB L2 cache.

5.1.2 Prefetching Performance

As execution time improvements due to prefetching often come at the expense of increased bandwidth usage, we evaluate performance along both metrics. Figure 5 shows the relative performance with G/DC, PC/DC, and PC/DC/MG for prefetch degree 16 and for L2 sizes of 256KB and 2MB. The figure also shows on the right vertical axis the memory bus traffic with respect to the traffic with no prefetch. For reference we show the relative performance with no prefetching (i.e., degree 0). Performance is relative to that of a system with an ideal L2 cache.

From this figure we see that for the majority of the applications there is a clear benefit from using a prefetcher. As expected, the gains from prefetching are higher with the smaller cache size of 256KB, but even with the fairly large cache size of 2MB the applications with low relative IPC also show significant benefits from prefetching. However, with the 256KB cache size for three applications prefetching with one or more of the schemes leads to a performance degradation: *gromacs* and *sjeng* with G/DC and *omnetpp* with all three prefetching schemes.

Comparing the three prefetching schemes, it is clear that no scheme performs best for all applications. The following analysis is for a 256KB L2 cache. Comparing G/DC and PC/DC reveals that PC/DC outperforms G/DC in 5 of the applications – by as much as 28% in *milc* – G/DC outperforms PC/DC in 9 of the applications – by as much as 59% in *lbm* – and they perform similarly (within 1% of each other) in the remaining 6 applications. On the other hand, comparing G/DC and our proposed prefetcher PC/DC/MG reveals that PC/DC/MG outperforms G/DC in 11 of the applications – by as much as 45% in *milc* – G/DC outperforms PC/DC/MG in 5 of the applications – by as much as 32% in *lbm* – and they perform similarly (within 1% of each other) in the remaining 4 applications. The reason why G/DC performs best in some cases is the existence of spatial (delta) correlation across misses generated by different static memory access instructions, which leads to better coverage, accuracy, or both, compared to the PC localized prefetchers. As expected, PC/DC and PC/DC/MG tend to outperform or underperform G/DC in the same applications, although in some cases PC/DC/MG outperforms G/DC even when PC/DC does not. Nevertheless, when G/DC outperforms both PC/DC and PC/DC/MG, the gap with the latter is often smaller than with the former. Finally, comparing PC/DC and PC/DC/MG reveals that PC/DC/MG outperforms PC/DC in 14 applications – by as much as 55% in *libquantum* – PC/DC outperforms PC/DC/MG in 4 applications – by as much as only 2% in *hmmer* – and they perform similarly (within 1% of each other) in the remaining 2 applications. Overall, this figure seems to indicate that PC/DC/MG shows more consistent performance across the entire range of applications than either PC/DC or G/DC. It also shows that between PC/DC and PC/DC/MG the latter seems the best performer overall. Results with the larger 2MB L2 cache show similar trends although both the relative benefits of prefetching and the relative benefits of each scheme are smaller than with the 256KB L2 cache.

Considering memory traffic, we see that the increase is often relatively small (less than 5%). However, in a few applications the traffic increase is significant. The largest increase is observed with *omnetpp* with increases of 108% and 123% with PC/DC and PC/DC/MG; and with a small increase of only 4% with G/DC. Despite this case, however, the traffic increase with G/DC is often larger, with the most notable cases being *gromacs* – with an increase of 73% with G/DC versus only 3% and 4% with PC/DC and PC/DC/MG – and *h264ref* – with an increase of 61% with G/DC versus 21% and 29% with PC/DC and PC/DC/MG. Finally, we note that the relative traffic increases with 2MB L2 cache are similar to those with the smaller 256KB L2 cache, although in absolute numbers the traffic with the larger cache are, obviously, smaller.

In the remaining sections we analyze the prefetching behavior in more detail in order to identify the reasons of PC/DC/MG's better performance.

5.2 Cache Miss and Prefetch Characterization

5.2.1 Prefetch Coverage

To gain some insight into the behavioral differences between G/DC, PC/DC, and PC/DC/MG we compute each prefetcher's coverage, defined as the ratio of the number of used prefetches over the total number of misses without prefetching. Figure 6 shows the coverage versus the prefetching degree.

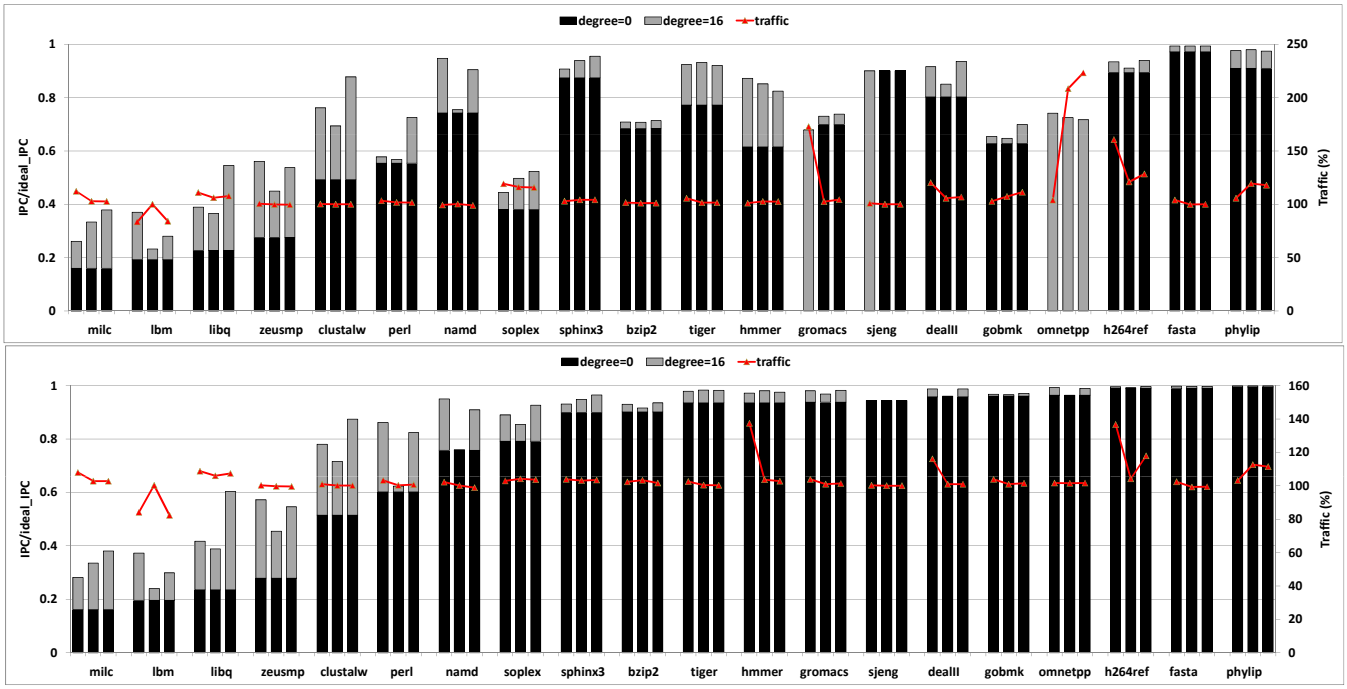

Figure 5: Relative performance (bars) and memory traffic (line) for prefetching degree of 16 with G/DC (left bar), PC/DC (middle bar), and PC/DC/MG (right bar), for 256KB L2 cache (top chart) and 2MB L2 cache (bottom chart).

From this figure we can see that the highest coverage varies across applications, but PC/DC/MG often offers the highest or close to highest coverage. Comparing PC/DC and PC/DC/MG, we see that the coverage of PC/DC/MG is often higher. This increased coverage can be attributed to two factors: one is the benefits of cross-stream correlations with the miss graphs and another is the improved timeliness, which means that prefetched lines are more likely to still be in the cache when needed (Section 5.2.2). For a few applications G/DC has higher coverage than either PC/DC or PC/DC/MG. This is both because by localizing the global miss stream PC/DC and PC/DC/MG may lose some opportunities to establish cross-stream correlations and because by having to observe enough misses for each stream separately PC/DC and PC/DC/MG may incur increased learning time to establish correlations. However, as will be seen in Section 5.2.2, this increased coverage almost invariably comes at the expense of much decreased prefetch accuracy.

5.2.2 *Prefetch Accuracy*

To gain further insight we compute each prefetcher's accuracy, defined as the ratio of the number of used prefetches over the total number of prefetches issued. Figure 7 shows the accuracy versus the prefetching degree.

From this figure we can see that in the vast majority of the cases the accuracy of PC/DC/MG prefetches is significantly higher than that of PC/DC. This difference is specially pronounced for higher prefetch degrees, which indicates that indeed PC/DC's deep prefetches tend to have lower accuracy and lead to wasted bandwidth usage. In fact, the accuracy of PC/DC/MG tends to remain stable or decrease only slightly with increasing prefetch degree while the accuracy of PC/DC tends to decrease rapidly.

Since PC/DC/MG issues prefetches in a more timely manner, the chances of these prefetches being actually used are higher than in PC/DC. In PC/DC accurate prefetches might be brought in to the cache too early and risk being evicted by some other data needed before them. PC/DC/MG tries to shorten the timespan between loading prefetched data in the cache and using it. In fact, after further investigation we found that often with larger prefetch degrees with PC/DC the fraction of prefetched lines displaced (before being used) by *demand misses* increases, which indicates that PC/DC's deep prefetches are not only being displaced by other prefetches but by demand misses that appear inbetween the prefetch request and its potential use. This, in turn, suggests that indeed such prefetch requests are being issued too early.

Comparatively, G/DC's accuracy is consistently lower than that of both PC/DC and PC/DC/MG, which can be linked to the often higher unpredictability of the global miss stream. Together, the coverage and accuracy numbers mostly explain the performance differences observed in Section 5.1.2. For instance, G/DC performs best for *lbm* because its coverage is much higher (with prefetch degree 16) despite its lower accuracy. On the other hand, in the case of *milc* the coverages are similar so the better performance of PC/DC/MG is mainly due to its better accuracy.

5.2.3 *Next-Stream Prediction Accuracy*

In addition to final prefetch accuracy (Section 5.2.2) it is useful to know the accuracy of PC/DC/MG in predicting the next stream (i.e., memory access instruction) to generate a miss. Figure 8 shows how often PC/DC/MG's prediction of next stream to miss is accurate when considering the next w misses.

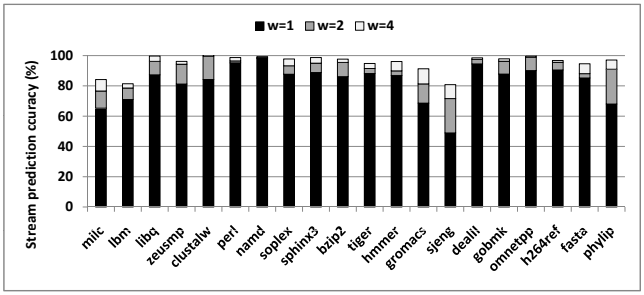

Figure 8: PC/DC/MG's accuracy in predicting the next PC to appear in the next w misses, for 256KB L2 cache.

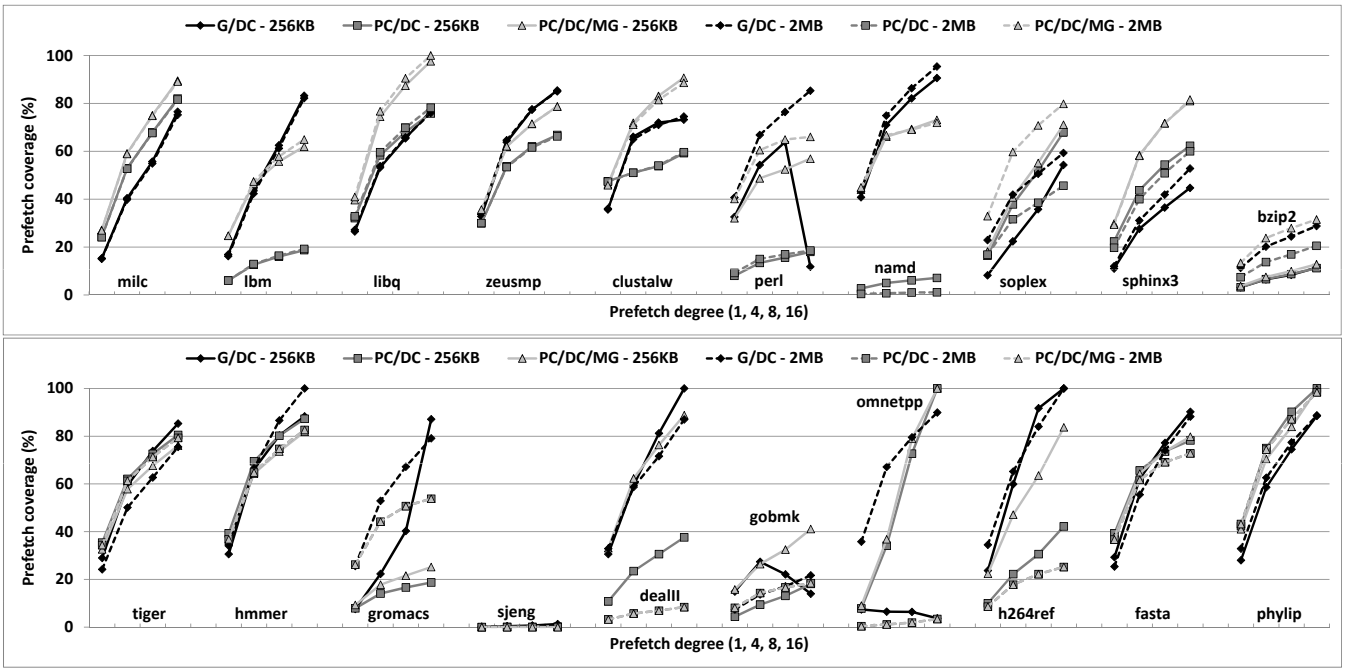

Figure 6: Prefetch coverage for varying prefetch degree with G/DC, PC/DC, and PC/DC/MG.

From this figure we can see that next-stream prediction accuracies are usually very high, even for the strict case of $w = 1$. With larger values of w accuracy is even higher, indicating that even when the predicted next stream does not correspond to the exact next miss it still appears within the next few misses. This shows that the miss graph approach described in Section 3.2, while simple, does capture the most frequent flow of instructions that miss.

5.2.4 Miss Distances

Another characteristic that helps explain the behavior of the three prefetch schemes is the distances between consecutive misses. Using an execution *without* prefetching, we measure the number of cycles between consecutive misses both coming from the same instruction and from any instruction. Figure 9 shows the miss distances grouped by ranges of number of processor cycles.

From this figure we can see that often a large fraction of L2 misses occur thousands or even hundreds of thousands of cycles apart. Moreover, the fraction of such distant misses often increases with cache sizes. More importantly, the fraction of such distant misses is significantly larger when we consider distances between misses from the same memory access instruction. Again, these results suggest that PC/DC with large degrees of prefetching, while still being able to eliminate some more misses, may issue such deep prefetches too early. This and the fact that many of such early prefetches are likely to be displaced before being used explain why PC/DC's accuracy is low for deep prefetching (Section 5.2.2) leading to unused prefetches and wasted bandwidth usage (Section 5.1.2). Like PC/DC/MG, G/DC does not suffer as much from such premature prefetches, but its much lower accuracy (Section 5.2.2) compensates for this and leads to a lower prefetch hit ratio.

The following section attempts to show some more insight into the behavior of stream chains.

5.3 Miss Graph Characterization

To gain more insight into the behavior of stream chaining, we attempt to characterize the miss graphs obtained following the graph construction described in Section 3.2.1. For this, we take snapshots of the IT at intervals of 1000 prefetch events (we also evaluated snapshots at intervals of L2 accesses and the results were similar). At each snapshot we obtained the complete miss graph constructed with the `NextIT` pointers. However, only links with `Ctr` above the minimum threshold of 3 were considered. In our experiments we found out that the miss graph constructed this way is in fact a collection of several connected components (CC). Nodes in the miss graph in one snapshot can be directly matched to nodes in other snapshots based on their PC value.

One important parameter for the success of the miss graph approach to prefetching is how stable the observed graphs are, i.e., how often the CC repeat themselves in successive snapshots. Unstable graphs will lead to poor next stream prediction accuracy. We attempt to quantify this stability by comparing the CC subgraphs seen at different snapshots. Because comparing the CC subgraphs across a very large number of snapshots is prohibitively expensive, we only compare the CC subgraphs of a snapshot against the CC subgraphs in the following window of 30 snapshots. We classify every CC subgraph at all snapshots as either *similar* to at least one other CC in one other snapshot in the window, or *unique*, meaning that there is no other similar CC in the window. We say that two subgraphs X and Y are similar if X is a subgraph of Y and X's nodes correspond to no less than 75% of Y's nodes, or if Y is a subgraph of X and Y's nodes correspond to no less than 75% of X's nodes. Note that with this definition, if two subgraphs are exactly the same they are also classed as "similar".

Table 2a shows – in columns 2 and 3 – the fraction of unique CC subgraphs for the two different cache sizes we consider. From this table we can see that indeed the number of unique subgraphs is very small for most benchmarks, although some benchmarks do have a significant fraction of unique subgraphs. To attempt to assess how the subgraph similarity holds across execution time, we varied both the window size and the number of prefetch events per interval (results not shown). The results in these cases did not vary noticeably. This indicates that subgraphs stay similar across large spans of time. To see why, consider both a case with smaller interval and a case with a larger window: if the fraction of unique subgraphs were to increase in either case with respect to the baseline

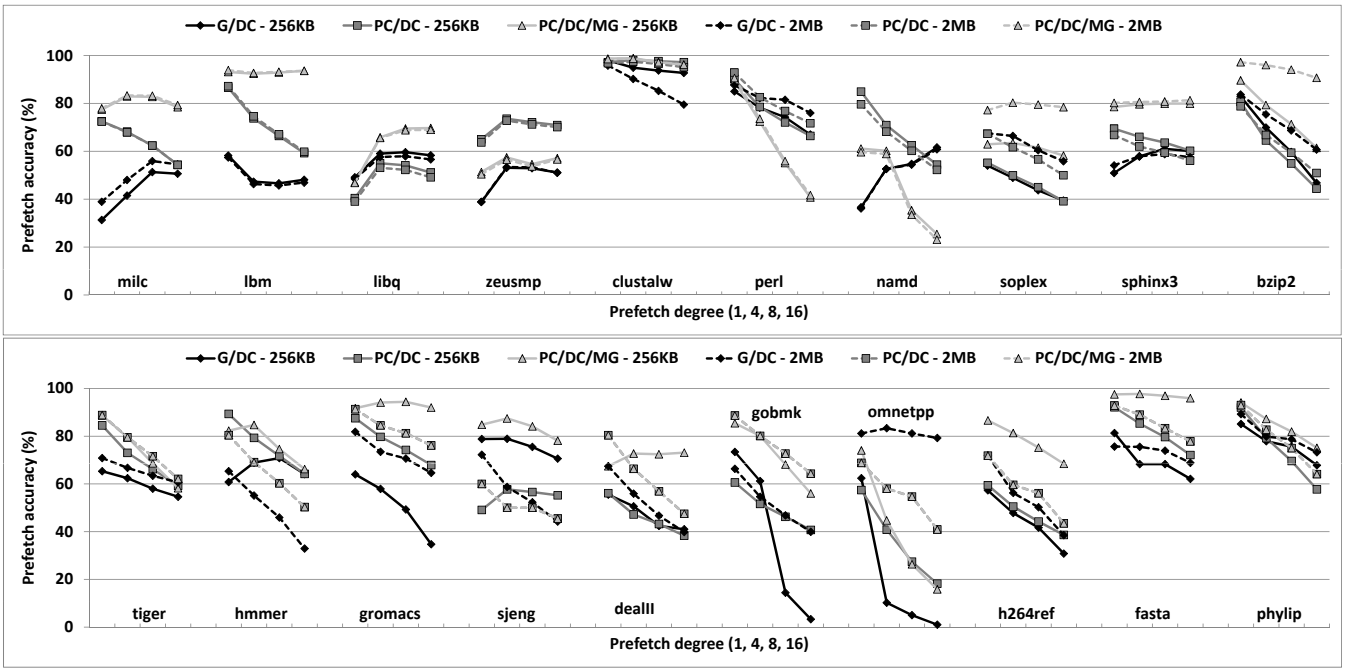

Figure 7: Prefetch accuracy for varying prefetch degree with G/DC, PC/DC, and PC/DC/MG.

interval and window sizes, it would be an indication that subgraphs do change frequently. However, this does not seem to be the case.

Table 2a also shows – in columns 4 to 7 – the average and range of number of nodes both per snapshot and per CC. We can see that the number of nodes per snapshot and, specially, per CC is not so large that managing the graphs – and, thus, operating the stream chaining mechanism – becomes too expensive, but in some cases it can contain enough nodes to make cycles long enough that visits to the same node (i.e., memory access instruction) are far apart in time, which partially explain the results in Section 5.2.4. Table 2b shows the average number of GHB lookups ("hops") required by each prefetcher to establish a delta correlation on a miss event, for a prefetch degree of 16. For PC/DC/MG the table shows the average number of hops in total for all streams considered in a miss event. From this table we can see that although the average total number of hops is larger in PC/DC/MG than in PC/DC, it is in all but two cases (*zeusmp* and *namd*) considerably smaller than in G/DC [3]. The reason for this is that the expected number of streams is small (Table 2a) and the number of hops per stream (results not shown) are often similar to those for PC/DC. On the other hand, the high average number of hops for G/DC can be explained by the higher unpredictability of a global miss stream.

6. RELATED WORK

Hardware prefetching is an established technique and is used in most commercial processors nowadays [3, 11]. Such prefetchers, however, are usually limited to relatively simple stride prefetchers [2, 4, 7] and address correlation prefetchers.

Address correlation using Markov chains was first proposed in [8] and delta correlation was first proposed in [9] for TLB prefetching. The adaptation of delta correlation to data prefetching was done in [12], where the PC/DC prefetcher was proposed. That work also proposed the GHB structure used in our work and introduced the taxonomy and notation for 2-level prefetchers that we extend here. The work in [6] proposed correlating cache line tags instead of complete addresses. The work in [14] proposed address localization using memory regions (called CZones) and with simple address stride correlation. This was extended in [13], which proposed a prefetcher with address localization with memory regions but with delta correlations and with dynamic tuning of region sizes and prefetch degree. The work in [19] proposed a similar address localization using memory regions, but in the context of a shared-memory multiprocessor, and with an exact bit-map encoding of addresses instead of delta correlations. An advantage of such address based localization approaches is that they do not require the PC of the missing memory access instruction, which may be difficult to obtain at the site of the prefetcher. However, they do not directly manage the temporal locality of misses as PC localization approaches do, but only indirectly in as much as there is or not temporal locality in the use of data. The work in [10] proposed localizing streams based on a history of the last few memory access instructions. However, to have an accurate history it tracks all memory access instructions, including those that hit in the cache, which puts greater demand on the prefetch engine and requires fairly large tables. The work in [21] proposed using the PC of branches to localize streams. It requires tracking branches as well as memory access instructions, which may not always be feasible.

Closer to our work, the idea of grouping misses that occur in chronological sequence into temporal streams was first used in [22]. Different from our work, that work was done in the context of shared-memory programs and focused on coherence misses. Thus, it exploited the fact that the same groups of shared data tend to be communicated in unison.

Concurrently with our work, [20] also exploits temporal and spatial correlation (and also PC correlation as part of the spatial component) and attempts to improve timeliness by reconstructing the total miss order from partial representations. The mechanisms used, however, differ from ours and consist of two parts: an extension of temporal streaming [22] to recurring spatially correlated blocks of misses [19], instead of individual misses, and a mechanism to reconstruct the miss order by recording miss distances inside a spatial stream.

[3] The larger average number of hops for PC/DC/MG compared to G/DC for *namd* reflects the former's difficulties to track correlations in this particular case and its consequent significant drop in accuracy (Figure 7).

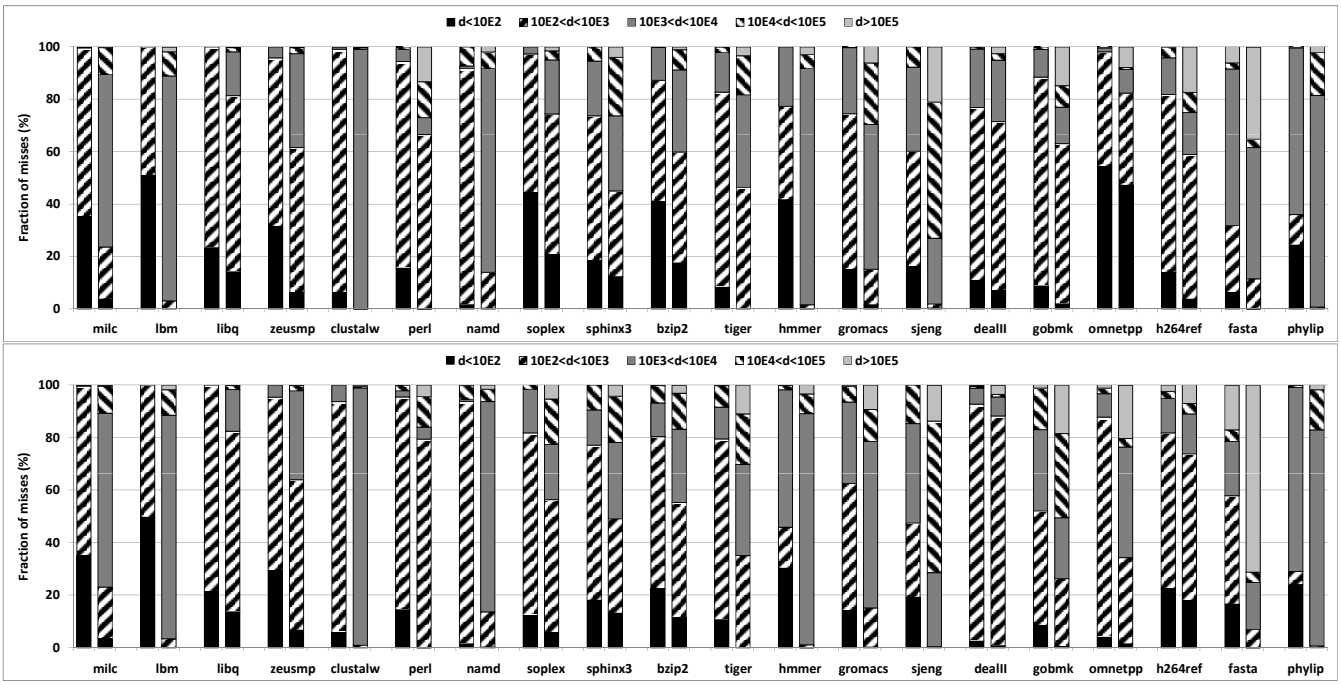

Figure 9: Miss distance for 256KB (top chart) and 2MB (bottom chart) L2 cache. For each benchmark the left bar is for any consecutive misses and the right bar is for consecutive misses caused by the same memory access instruction.

Mostly orthogonal to the issues of how to localize streams and correlate addresses, [10] and [5] proposed using a dead-block predictor to identify replaceable cache lines and trigger prefetches. Prefetchers based on dead-block prediction can mitigate the problem caused by early prefetched lines displacing useful data in the cache. However, they do not address the problem that such early prefetched lines may be themselves evicted before being used and having to be re-fetched. Also orthogonal to our work, [18] proposed using a user-level thread running in a processor-in-memory configuration to execute the prefetching algorithms.

Also in the area of software prefetching, recent works have focused on exploiting temporal and spatial correlation in the miss address stream. Most notably, [23] proposes a dynamic optimization framework that combines temporal prefetching with object-based spatial prefetching and is able to adaptively tune prefetch distances to improve timeliness.

7. CONCLUSIONS

In the context of data prefetching, separating the history of misses into multiple streams – for instance according to the PC of the memory access instructions causing the misses – has been shown to achieve improved performance because it allows the prefetcher to focus on groups that show predictability and discard random and less predictable accesses. Localization offers a second level of design choice on top of the choice of mechanism for correlating the addresses within a stream. The problem with localizing streams is that important sequencing information of the misses is lost. In this paper we proposed adding a third level to the operation of prefetchers that allows them to link various localized streams into predictable chains of memory access instructions – we call this *stream chaining*. The key idea of stream chaining is to *partially* reconstruct the sequencing information in the miss stream, such that the result corresponds directly to the *common path of misses* followed by the application. With streams localized by memory access instructions' PC's, stream chaining allows a memory access instruction to trigger prefetches not only for itself, but also for following different static memory access instructions, whose addresses have been grouped in a different stream. In this way the prefetcher is aware of its context, as in memory access instruction PC based localization schemes, and is not limited to prefetching deeply for a single memory access instruction but can instead adaptively prefetch for other memory access instructions closer in time. We presented an initial scheme for stream chaining, which we call *miss graph prefetching*, that is simple, captures well common application behavior, and can be easily implemented on a simple extension to the Global History Buffer (GHB). We evaluated the proposed prefetcher with benchmarks from the SPEC 2006 and BioBench suites and compared it against a state-of-the-art memory access instruction PC based localization prefetcher – PC/DC [12]. Experimental results showed that the proposed prefetcher consistently achieves better performance than PC/DC – it is only outperformed in very few cases and then by only 2%, and it outperforms PC/DC by as much as 55%, with an average improvement of 10%. Our detailed experiments showed that this performance gain comes from a higher accuracy, specially for high prefetch degrees, which leads to a larger fraction of prefetched lines being actually used before being evicted.

This paper represents initial work on a novel class of prefetchers and, thus, there are many possible avenues for future work. One extension of this work is to investigate alternative ways of chaining streams (Section 3.2.1) and/or alternative prefetch policies based on the resulting graphs (Section 3.2.2). For example, an adaptive mechanism to control how much to prefetch on each graph node could be easily added to PC/DC/MG. A more comprehensive extension is to investigate the stream chaining idea with stream localization approaches other than memory access instruction PC based, such as with address region localization [13].

8. REFERENCES

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. "BioBench: A benchmark suite

Benchmark	Unique Subgraphs (%)		Nodes			
			Snapshot		CC	
	256KB	2MB	256KB	2MB	256KB	2MB
milc	4.7	3.0	[2, 15] 7.7	[2, 15] 7.9	[1, 7] 3.6	[1, 7] 3.5
lbm	22	12	[2, 20] 7.9	[2, 18] 6.7	[2, 18] 3.7	[2, 18] 4.2
libq	0.8	1.3	[2, 23] 19	[1, 24] 21	[1, 18] 7.0	[1, 18] 7.4
zeusmp	11	10	[2, 18] 11	[2, 15] 8.8	[2, 9] 4.4	[2, 10] 4.1
clustalw	1.1	7.2	[3, 10] 9.3	[2, 9] 6.7	[2, 10] 8.2	[1, 9] 6.6
perl	11	7.1	[1, 16] 8.6	[2, 18] 9.9	[1, 9] 3.3	[1, 9] 3.6
namd	21	23	[2, 8] 5.8	[2, 8] 6.0	[2, 8] 5.0	[2, 8] 5.8
soplex	2.8	12	[1, 30] 12	[1, 11] 6.2	[1, 10] 3.6	[1, 6] 3.0
sphinx3	11	9.6	[4, 16] 13	[2, 11] 7.1	[1, 15] 5.7	[1, 5] 3.4
bzip2	5.6	15	[1, 38] 20	[1, 25] 13	[1, 9] 3.8	[1, 7] 3.7
tiger	5.4	6.7	[7, 41] 30	[6, 34] 23	[1, 18] 4.2	[1, 14] 3.8
hmmer	12	9.4	[15, 50] 38	[13, 36] 28	[1, 33] 5.4	[1, 26] 4.5
gromacs	15	42	[2, 25] 13	[4, 13] 9.8	[1, 12] 3.6	[1, 7] 4.4
sjeng	45	29	[2, 13] 7.7	[2, 14] 8.4	[2, 13] 5.1	[2, 12] 5.2
dealII	6.4	43	[1, 25] 14	[1, 9] 4.7	[1, 11] 4.1	[1, 7] 3.1
gobmk	20	31	[1, 10] 5.2	[2, 13] 9.3	[1, 5] 3.4	[1, 9] 4.5
omnetpp	6.2	1.6	[1, 9] 3.7	[3, 14] 4.8	[1, 4] 3.0	[1, 3] 1.8
h264ref	14	22	[1, 7] 4.6	[1, 9] 8.0	[1, 4] 2.9	[1, 4] 2.8
fasta	8.9	8.9	[8, 19] 15	[8, 17] 15	[1, 16] 5.4	[1, 9] 3.9
phylip	13	37	[8, 36] 26	[8, 14] 11	[1, 24] 5.1	[1, 10] 4.6

(a)

GHB hops					
G/DC		PC/DC		PC/DC/MG	
256KB	2MB	256KB	2MB	256KB	2MB
457	452	43	43	55	55
286	297	7.7	8.8	9.2	11
421	429	52	57	49	51
346	342	31	34	371	384
235	235	9.5	11.6	31	31
194	375	19	25	94	143
190	176	37	38	353	352
472	453	79	55	101	60
484	472	118	131	123	137
507	496	202	104	215	109
371	415	27	31	109	60
254	269	19	57	131	158
480	401	76	44	86	70
511	511	64	95	148	332
357	311	85	94	116	97
390	496	124	106	136	116
434	301	306	55	313	67
383	317	18	47	30	63
327	316	24	12	28	14
383	355	52	49	76	44

(b)

Table 2: (a) Statistics related to miss graphs: Unique refers to percentage of unique connected components, Snapshot refers to the range and average number of nodes per snapshot, and CC refers to the range and average number of nodes per connected component. (b) Average number of "hops" required in the GHB structure to establish correlation(s) on a miss event, for prefetch degree 16.

of bioinformatics applications." *Intl. Symp. on Performance Analysis of Systems and Software,* pages 2-9, March 2005.

[2] J.-L. Baer and T. F. Chen. "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty." *Supercomputing Conf.,* pages 176-186, November 1991.

[3] J. Doweck. "Inside Intel Core Microarchitecture and Smart Memory Access." White paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[4] J. W. C. Fu, J. H. Patel, and B. L. Janssens. "Stride Directed Prefetching in Scalar Processors." *Intl. Symp. on Microarchitecture,* pages 102-110, December 1992.

[5] Z. Hu, S. Kaxiras, and M. Martonosi. "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior." *Intl. Symp. on Computer Architecture,* pages 209-220, May 2002.

[6] Z. Hu, M. Martonosi, and S. Kaxiras. "TCP: Tag Correlating Prefetchers." *Intl. Symp. on High Performance Computer Architecture,* pages 317-326, February 2003.

[7] N. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." *Intl. Symp. on Computer Architecture,* pages 364-373, May 1990.

[8] D. Joseph and D. Grunwald. "Prefetching Using Markov Predictors." *Intl. Symp. on Computer Architecture,* pages 252-263, June 1997.

[9] G. B. Kandiraju and A. Sivasubramaniam. "Going the Distance for TLB Prefetching: An Application-Driven Study." *Intl. Symp. on Computer Architecture,* pages 195-206, May 2002.

[10] A.-C. Lai, C. Fide, and B. Falsafi. "Dead-Block Prediction & Dead-Block Correlating Prefetchers." *Intl. Symp. on Computer Architecture,* pages 144-154, June 2001.

[11] H. Q. Le, W. J. Starke, J. Stephen Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. "IBM POWER6 Microarchitecture." *IBM Journal of Research and Development,* vol. 51, no. 6, pages 639-662, November 2007.

[12] K. J. Nesbit and J. E. Smith. "Data Cache Prefetching Using a Global History Buffer." *Intl. Symp. on High Performance Computer Architecture,* pages 96-106, February 2004.

[13] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. "AC/DC: An Adaptive Data Cache Prefetcher." *Intl. Conf. on Parallel Architectures and Compilation Techniques,* pages 135-145, September 2004.

[14] S. Palacharla and R. E. Kessler. "Evaluating Stream Buffers as a Secondary Cache Replacement." *Intl. Symp. on Computer Architecture,* pages 24-33, May 1994.

[15] D. G. Pérez, G. Mouchard, and O. Temam. "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms." *Intl. Symp. on Microarchitecture,* pages 43-54, December 2004.

[16] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator. http://sesc.sf.net

[17] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Research Report, 2001/2.

[18] Y. Solihin, J. Lee, and J. Torrellas. "Using a User-Level Memory Thread for Correlation Prefetching." *Intl. Symp. on Computer Architecture,* pages 171-182, May 2002.

[19] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. "Spatial Memory Streaming." *Intl. Symp. on Computer Architecture,* pages 252-263, June 2006.

[20] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. "Spatio-Temporal Memory Streaming." *Intl. Symp. on Computer Architecture,* June 2009.

[21] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. "Branch History Guided Instruction Prefetching." *Intl. Symp. on High Performance Computer Architecture,* pages 291-300, January 2001.

[22] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. "Temporal Streaming of Shared Memory." *Intl. Symp. on Computer Architecture,* pages 222-233, June 2005.

[23] W. Zhang, B. Calder, and D. M. Tullsen. "A Self-Repairing Prefetcher in an Event-Driven Optimization Framework." *Intl. Symp. on Code Generation and Optimization,* pages 50-64, March 2006.