Modeling Memory Location Predictors for Pointer-Based Codes

Antoliano Dávila, Eladio Gutiérrez, Oscar Plata, and Emilio L. Zapata

Departament of Computer Architecture
University of Malaga, Malaga, Spain,
tolian@ac.uma.es

Abstract. It is unknown, at compile time, the locations in the heap where the nodes of a linked data structure shall be allocated during the execution of a dynamic code. This lack of information limits our ability of exploiting the reference locality when such data structures are created, traversed or modified.
For pointer-based codes, where the heap memory locations that will be referenced in the near future are unknown, the performance efficiency of the prefetcher greatly depends on the accuracy of the memory location prediction techniques. For this reason, many predictors in the context of irregular and pointer-based applications have been proposed in the literature. Unfortunately, each one of these predictors performs well only for certain data access patterns while it performs poor for others.
In this work we propose a model to parameterize a wide selection of prediction methods in the context of pointer-based codes. With this model we intend to state a baseline to compare the performance of these predictors and determine in which conditions (code and data properties) each predictor works the best. In the context of this model, we present an experimental evaluation of the behavior of several predictors on a representative set of pointer-intensive codes extracted from the Olden suite. These results show the usefulness of our model as a tool to conduct unbiased comparisons of such predictors.

1 Introduction

During the past decades we have been noticing an steady increase in the processor-memory latency gap, either in mono and multiprocessor architectures. This fact causes an important performance problem for those applications that do not exhibit enough temporal and/or spatial data locality to take full advantage of the cache hierarchy. This is the case of pointer-intensive applications, where a large extent of memory accesses is generated through indirections (pointers). In this kind of codes the pointer-based data structures are allocated, modified and deallocated during the execution. Components of these data structures are stored in the heap in locations determined at runtime. Furthermore these locations vary between different executions and even during the same execution (for example, via garbage collection operations or re-allocations) [7, 10]. In addition, due to allocation/deallocation operations, the same heap location may

be reused by different data components in different execution phases. All these features make that determining the amount of locality these applications hold and how to improve it be a hard task for the compiler. In fact, in many cases this task becomes impossible at compile time [23].

Since locality exploitation is seriously limited at compile level for pointer-based applications, an alternative to improve performance is hiding the memory latency using techniques like prefetching, one of the most important. A prefetcher tries to issue fetches to missed data before the processor demands it. Two main issues have to be considered in a prefetch method [28]: the prediction method and the promptness of the prefetch. If the accuracy of the predictor is not good enough, the effectiveness of the prefectching is limited due to an increase of memory traffic and cache pollution. The promptness of the prefetch means that it cannot be issued too early (as cache becomes polluted) nor too late (as latency is not completely hidden). In this work we focus only on the first issue, the prediction method.

In the context of pointer-based applications, it is very difficult for the compiler to determine which data locations will be referenced in the near future when executing the program (accuracy of the prediction). That means that a precise data location prediction is needed for an effective prefetching. Having this fact in mind, many researchers have proposed in the literature different prediction methods for pointer-based codes. In general, these predictors obtain variable performance, very dependent on the memory access patterns of the program.

Many approaches for data prefetching have been proposed in the literature. Some of them have been designed to be completely implemented in hardware [24, 2, 33, 17, 34, 26, 14, 9], having the advantages of being performed at run-time, being transparent to the programmer, not introducing explicit execution overhead on the program by prefetch instruction addition, and requiring no code transformations. Others are software solutions [4, 16, 6, 29, 20, 31, 3], allowing a larger analysis scope, adding no complexity to the processor and allowing to implement sophisticated optimization strategies. Finally, a third class of methods are hybrid solutions [25, 1, 30, 32], combining both hardware and software approaches in order to reduce hardware requirements and adjust to dynamic behavior.

The contribution of this paper is a characterization of a representative set of data location predictors for pointer-intensive programs. As a result of this analysis, a basic set of quantitative parameters are proposed in order to characterize the prediction methods, and establish a baseline that permit to compare their behaviour under several locality conditions. An experimental evaluation of several prediction methods in this context is presented using benchmark codes extracted from the Olden suite [5]. Such experiments show the behavior of predictors on the memory address stream derived from the access to the heap of the structures dynamically allocated.

The rest of paper is organizated as follows. Section 2 presents a general description of the prediction process in the context of data prefetching, reviews a selection of several prediction methods developed for pointer-based codes, and introduces a parameterized characterization of the prediction techniques. Sec-

tion 3 shows some experimental results using Olden benchmarks, allowing to compare the quality of the prediction methods. Finally, we briefly present some conclusions of this work.

2 Charaterization of Prediction Methods

When analyzing the effectivity of a prefetching technique, one of the most important aspects we have to consider is the prediction method. We can distinguish, at least, two key features of the predictor that have a main influence in such effectivity: the prediction effort and the prediction quality. The prediction effort corresponds to the cost of accomplishing the process of prediction from a specific input data set and an internal state. This cost is associated to the implementation of the method. The prediction quality is the capacity of the method to generate a useful result, that is, something that will really occur in the future.

Usually there is a trade-off between these prediction features and the prefetch effectivity. With a larger prediction effort a better quality is expected. Nevertheless, this large effort could imply a higher time overhead that may cause a prefetching effectivity loss because the resulting prediction is obtained too late (promptness is not fulfilled). On the other hand, it can be thought that low-effort predictors may exhibit a lower time overhead, but probably the quality is also reduced polluting the cache content with wrong data (misspredictions).

The prediction effort is a feature directly related to the implementation of the method, either hardware and/or software. Therefore, it can be important to define a set of quantitative characteristics with the aim of dimensioning the implementation of the predictor and, consequently, of the prediction effort. For example, in the case of a hardware predictor a possible characteristic could be the chip area it occupies in relation to the cache level for which predictions are made. In addition, such a characterization may be useful to establish a reference to compare different prediction methods. In general, this characterization is associated to a prediction model. Next a model of this kind will be introduced trying to cover a wide range of prediction methods used in the context of pointer-based applications.

A flowchart for the prediction process is outlined in Fig. 1. Basically starting from the execution of the code, a memory location trace is taken as input for the prediction system. Such an execution can be in real time, for example if it is oriented toward a hardware or software prefetching, or it can be deferred, for example in order to profile the code for further instrumentation or optimization. The memory location trace can be interpreted commonly either as a stream of absolute memory addresses or in a differential mode (strides or differences between consecutive address requests). Note that absolute locations will give information about possible repetitions in the stream, whereas the information concerning differences or strides is more related with changes or transitions in the program behaviour.

Often it makes no much sense to accomplish the prediction for the complete memory trace, but only for a subset of the memory requests (filtering of the

memory stream). One of the reasons is because using all memory requests in the program may cause wrong predictions, as some memory addresses are not needed to be predicted but may act as noise polluting the predicted stream. In addition, filtering the memory stream implies dealing with a lower amount of data, allowing to design lighter predictors (although an excessive filtering may compromise prediction quality). Another reason to filter the memory stream is the low latency of the memory hierarchy levels closer to the processor, specially, the L1 cache. Predictions for these levels may be generated too late. In the case of the programs analyzed in this paper, we applied two filters. On the one hand, all memory locations not included in the heap were filtered out (as we are interested in dynamic memory structures). On the other hand, all predictions were carried out on those memory request causing L1 misses.

Finally, the predictor usually requires some feedback mechanism, in order to adapt the predictor internal state to the dynamic nature of the program. Note that programs often are composed of several phases with different features in the memory access patterns. The feedback is used to introduce in the predictor information about the program behaviour in a recent past as well as information about the prediction quality at each moment (error degree of the prediction).

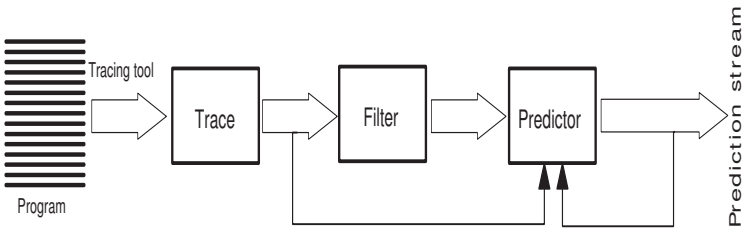

Fig. 1. Flowchart of prediction phases.

2.1 Selected Predictors

With the purpose of obtaining a set of common features, we have selected a representative number of predictions techniques that have been used in the context of pointer-based programs. Next these predictors are introduced and briefly described.

- *Pointer Cache (PC)* [11] is specifically targeted at speeding up recurrent pointer accesses. This cache only stores pointer transitions (between heap objects) so that we have a reduced representation of the important pointer transitions. The pointer cache is examinated as a prediction table to select the next value. This method is implemented as a memory cache where each entry stores the location of the initial heap object and the destination.

- *History Pointer (HP)* [19] is entirely software technique of data prefetching that generates prefetch instructions inserting them into the code. Basically this method inserts new jump-pointers (called history pointers) to other nodes in advance in the linked data structure, after the observation of a recent traversing.
- *Global History Buffer (GHB)* [21] is based on a buffer that holds the most recent values happened in the program. The prefetching structure has two levels: an index table and the global history buffer. The index table is accessed with a key as in conventional prefetch tables, where each entry in the table contains a pointer into the Global History Buffer. This global buffer is a FIFO table that holds the most recent values produced.
- *Markov predictors* are based on Markov models, being a well-known technique used in prefetching [15]. The Markov predictor captures the past activity of a program and uses this information to predict future references. The main element of a Markov-based predictor is a transition graph, where each node represents one state (past memory locations) of the model and each edge represents the probability of transition from one state to other one.
- *Spectral Prefetching (SP)* [27] derives from the tag correlating prefetching (TCP) [13], being able to capture the frequency information (harmonics) within a pattern in a memory location sequence. Also this approach is complemented with an adaptive mechanism that switches dynamically between an absolute or differential mode under certain conditions. The memory space is divided into different zones (TCzones), analogue to associative cache sets.
- *Greedy prefetching* is a technique that fetches all possible next nodes for a given node of a data structure [19]. The compiler inserts new instructions to prefetch all nodes. Note that if the connectivity of data structure is high this prefetching mechanism takes the risk of polluting the cache content.
- *ESODYP* [3] is a pure software approach based on a Markov model. It remembers sequences of strides between successive data accesses and tries to predict the next access when a new address is given to the predictor. For each element, the method stores a history of next accessed elements, and then it can predict certain number of elements in advance.
- *Sequitur* [22] is an algorithm that infers a hierarchical structure from a symbol sequence by replacing repeated phrases with a grammatical rule. It has been used to compress the trace and extract hot data streams [8] in order to generate and insert prefetching instruction into the code.
- *Dependence based prefetching (DBP)* [24] dynamically identifies load instructions that access the linked data structures, collecting these loads with their dependence relationships. It stores correlation information between a load instruction that produces an address and a subsequent load that uses this address (producer/consumer pairs).
- *Pointer data prefetch (PDP)* [17] identifies pointer loads instruction and the target value of pointer loads, using a small cache to save these locations dynamically and then prefetch potential base address of next nodes.

2.2 Prediction Characteristics

Common characteristics related to the dimensionality of different prediction methods can be stated. Such parameters will determine in part the prediction effort, as larger values of these parameters involve a higher cost or complexity. In the case of a hardware implementation this fact will result in higher needs of resources, and in the case of a software implementation a higher prediction time will make difficult the fulfillment of promptness requirements. The parameters we have chosen are the following:

- *Prefetch distance:* It is the number of locations in advance that the method try to predict. This parameter is strongly related with the ability of the method to hide latency.
- *Context:* This parameter is related to the number of past values used as a history to make the prediction. The size of this history is the context of prediction. A large context implies a better accuracy on predictions, but implies a lower number of predictions.
- *Width:* This parameter is the number of locations predicted in a single step, from which the method must select one or several to prefetch. A large width may cause problems of prediction pollution.
- *Learning size:* This parameter is the window of memory locations analyzed in the recent past used to build the context. Like in the context, a large learning size means more accurate predictions, but implies less predictions.
- *Predictor table size:* Most of the analyzed predictors use a set of tables to store the internal state. This parameter corresponds to the number of entries of such tables, affecting directly to the cost of implementation.

Prediction method	**Input data**	**Prefetch Distance**	**Width**	**Context**
PC	Heap addresses	1	1	1
HP	Heap addresses	Variable	1	1
GHB	Variable	Variable	Variable	1
Markov	L2 miss stream	1	1	1
SP	L2 miss stream	Variable	1	2
Greedy	Heap addresses	1	Variable	1
ESODYP	Strides	Variable	1	Variable
Sequitur	Data reference	Variable	1	Variable
DBP	Heap addresses	1	1	1
PDP	Heap addresses	1	1	1

Table 1. Characteristics of the prediction methods

In Table 1 the features of the analyzed predictors in terms of the described parameters are summarized. This information may be useful to establish a common base to compare and evaluate the methods in similar conditions. For each

method, the second column of the table shows the type of the elements of the input location stream after the filtering stage. The last three columns corresponds to the first three described parameters. The *learning size* and *prediction table size* parameters are not included because they are tunable in all cases.

In addition to the characteristics of the methods, properties of the memory access patterns also have a important influence in the prediction behaviour. This is specially significant for irregular and dynamic access patterns, as is the case of pointer-intensive applications like those we are considering in this work. A first type of features of memory access patterns may be considered static. These static features are associated with the definition of the linked data structures. Examples of such features are the next. The *FanOut* of a node measures the possible number of pointers to other nodes. A larger *FanOut* involves a higher connectivity, making harder to predict further ahead. Also the *node size* in memory and its *data type* will affect the prediction because predictors work with a specific level of memory granularity. A second type of memory access pattern features is dynamic, that is, properties that are unknown until runtime. Usually, these features are associated with changes in the shape of the data structure. For example, the *number of nodes*, the average number of pointers to other nodes (*PointsTo*) or the average number of pointer from other nodes (*PointsFrom*). Another feature is the *variability*, that measures in average how many times a pointer changes its value. It is expected poor prediction results for programs with a high variability.

3 Experimental results

A collection of prediction techniques has been evaluated experimentally over a representative set of pointer-intensive codes extracted from the Olden suite [5]. Experiments were carried out after obtaining the trace of memory locations by means of the instrumentation tool PIN [18]. Memory patterns have been processed in both absolute and differential (strides) ways and only those accesses pertaining to the heap (pointers) has been considered, as long as they are the basis for the dynamic data structures.

Two metrics are often used to evaluate the quality of a predictor. These are the *coverage* and the *accuracy*. Although they do not provide a complete measurement of the performance because they do not take into account information about the prediction effort, namely requirements or processing time, such metrics can be used as a good indicator of the usability for a given method. Derived from definitions found in [12], we will introduce these metrics as follows:

$$\text{accuracy} = \frac{\text{useful predictions}}{\text{total predictions generated}}$$

$$\text{coverage} = \frac{\text{prediction hits}}{\text{misses without predicts}}$$

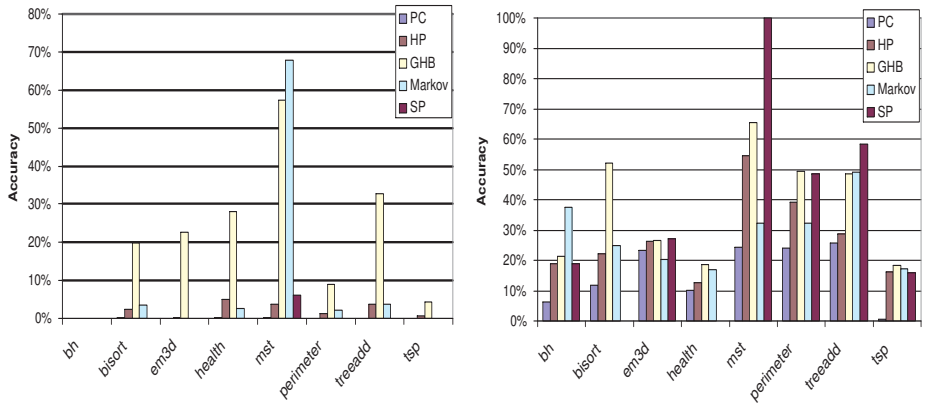

Fig. 2. Accuracy of prediction for Olden codes using absolute addresses (left) and strides (right) (line size=16B, prefetch distance=2 (absolute) / 1 (strides), context=1, width=1, table size=1024 entries, learning size=2048)

Figs. 2 and 3 show the accuracy of prediction for different prediction methods on a set of Olden programs using input location streams based on absolute memory addresses and strides. Fig. 2 corresponds to a particular conditions of the predictor characteristics, while in Fig. 3 the results are obtained fixing the cost of storing the internal state. Note that the dynamic range for strides is considerably lower than for absolute address and this make stride patterns more repetitive causing a better behavior of predictors in almost all the cases. However, we observe, in general, a great variability in the results over the prediction methods and the input programs. This fact shows either there is still ample room for improving the prediction techniques for pointer-based programs, or predicting in this context is a too difficult task. A possible solution to this problems could be a multi-prediction approach, where a specific predictor is used in different phases of the input location stream depending on some properties of them.

4 Conclusions

In general it is a hard task for the compiler to exploit reference locality when creating, traversing and modifying pointer-based data structures. In practice, we have usually to resort to runtime techniques and/or hardware support. With the aim of obtaining such locality features we have analyzed the behavior of several data reference prediction techniques applied to a representative set of pointer-intensive codes extracted from the Olden suite. We have accomplished an exploration of the prediction abilities of such techniques in terms of different design characteristics.

From our experimental results and evaluation it can be noted that there is no prediction technique that work well for all kind of reference patterns. One of the

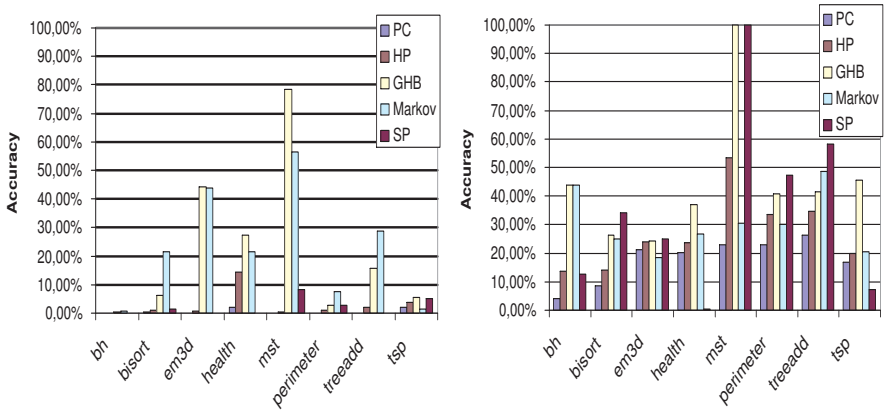

Fig. 3. Accuracy of prediction limiting the size of the tables storing the predictor internal state (to 32KB)

most important factors that negatively influences the prediction is the dynamic nature of the code. In addition, for all tested codes, there is none that exhibits locality features amenable for good results for all prediction techniques.

In general, it can be seen that most of the tested codes exhibit various phases in the heap memory access pattern, with different characteristics, that cause prediction difficulties for a single, uniform technique. There are also programs with the same dynamic data structure but with very different access patterns, influencing significantly the resulting predictions. Both, shape and traversing pattern, are essential for a good data location prediction.

References

1. Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. *ISCA'01: Proceedings of the 28th annual international symposium on Computer architecture*, 29(2), 2001.
2. Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. Correlated load-address predictors. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, 1999.
3. Jean Christophe Beyler and Phillipe Clauss. Esodyp: An enterely software and dynamic data prefetcher based on a markov model. *CPC'06: Compilers for Parallel Computers*, 2006.
4. Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.
5. Martin C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Princeton, NJ, USA, 1996.

6. Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *PLDI '01: Proceedings of the conference on Programming language design and implementation*, 2001.
7. Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.
8. Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.
9. Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Transaction on Computer Systems*, 22(2), 2004.
10. Jacques Cohen. Garbage collection of linked data structures. *ACM Comput. Surv.*, 13(3):341–367, 1981.
11. Jamison Collins, Suleyman Sair, Brad Calder, and Dean Tullsen. Pointer cache assisted prefetching. *MICRO-35: International Symposium on Microarchitecture*, 2002.
12. Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *SIGOPS Operating System Review*, 36(5):279–290, 2002.
13. Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. Tcp: Tag correlating prefetchers. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.
14. Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, 2004.
15. Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.
16. Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *HPCA'00: International Symposium on High Performance Computer Architecture*, 2000.
17. Shih-Chang Lai and Shih-Lien Lu. Hardware-based pointer data prefetcher. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, 2003.
18. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the conference on Programming language design and implementation*, 2005.
19. Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transaction on Computers*, 48(2):134–141, 1999.
20. Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P.Geoffrey Lowney, and Robert Cohn. Profile-guided post-link stride prefetching. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, 2002.
21. Kyle Nesbit and James Smith. Data cache prefetching using a global history buffer. *HPCA'04: High Performance Computer Architecture*, 2004.
22. Craig G. Nevill-Manning and Ian Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

23. Oscar Plata, Rafael Asenjo, Eladio Gutiérrez, Francisco Corbera, M.A. Navarro, and Emilio L. Zapata. On the parallelization of irregular and dynamic programs. *Parallel Computing*, 31(6):544–562, 2005.
24. Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.
25. Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA '99: Proceedings of the 26th annual International Symposium on Computer Architecture*, 1999.
26. Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transaction on Computers*, 52(3):260–276, 2003.
27. Saurabh Sharma, Jesse Bey, and Thomas Conte. Spectral prefetcher: An effective mechanism for l2 cache prefetching. *ACM Transactions on Architecture and Code Optimization*, 2(4):423–450, 2005.
28. Viji Srinivasan, Edward Davidson, and Gary Tyson. A prefetch taxonomy. *IEEE Transaction on Computers*, 53(2):126–140, 2004.
29. Artour Stoutchinin, José N. Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain, and Alban Douillet. Speculative prefetching of induction pointers. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 289–303, 2001.
30. Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, 2003.
31. Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *PLDI '02: Proceedings of the Conference on Programming language design and implementation*, 2002.
32. Chia-Lin Yang, Alvin R. Lebeck, Hung-Wei Tseng, and Chien-Hao Lee. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Transactions on Architecture and Code Optimization*, 1(4):445–475, 2004.
33. Lixin Zhang, Sally A. McKee, Wilson C. Hsieh, and John B. Carter. Pointer-based prefetching within the impulse adaptable memory controller: Initial results. 2000.
34. Youtao Zhang and Rajiv Gupta. Enabling partial cache line prefetching through data compression. *ICPP'03: International Conference on Parallel Processing*, 00, 2003.