TDT4260 Computer Architecture Mini-project: Investigating the Trade-offs of Markov Chains based prefetcher

***Abstract*—Over the last decade, several measures were proposed to overcome the huge memory gap. Prefetching is considered as one of the most powerful techniques to overcome this gap. Prefetching provides the data to the cache that microprocessor needs for calculations in advance, this results in a more efficient core and a higher throughout for the processor core. The focus of this paper is the implementation of Markov prefetcher as a good example of address-based prefetchers against data-based prefetching algorithms as DCPT and RPT. Several models of these algorithms were implemented to compare and evaluate the trade-offs of using one algorithm than the other. CPU2000 test-bench was used along with the M5-simulator to conduct the performance analysis. Specific metrics were included as speed-up, complexity, coverage, accuracy and overhead memory usage to compare between the several algorithms used. The results show higher leverage for our DCPT implementation in all of the tests over the other algorithms including Markov.**

***Index Terms*—RPT, DCPT, Markov, perceptron, data-based prefetchers, address-based prefetchers, m5-simulator, CFP2000.**

I. Introduction

MEMORY wall is defined as the gap between the processor speed and the memory bandwidth (1). Taking Moore's Law into account, processor speed had been so far the main focus of the microprocessor architectures (2). That and other factors lead to several solutions trying to address the gap between the processor speed and the memory speed. Prefetching was counted as a greatly useful measure in hiding this memory gap. In the Intel Xeon architecture (3; 4), 3 cache levels, alongside prefetching algorithms (both on hardware and software levels) were used. Ideally, this should eliminate nearly all cache misses. However, prefetching algorithms are not ideal. This is proven by the results of the recent surveys (5) that even while using prefetching algorithms, cache misses still happen. That explains the ongoing research on prefetching algorithms, that currently integrates machine learning algorithms to detect patterns in the processor memory access (6; 7).

In this paper, we discuss different data prefetching algorithm as reference prediction tables (RPT), Delta Correlation Prediction Pattern (DCPT). We also shed the light on some of the state of the art algorithms as perceptron prefetching which uses a machine learning algorithm to detect the access pattern (8; 9; 10). We extensively discuss the implementation of the Markov prefetching algorithm against all of the aforementioned algorithms in terms of complexity, memory size needed, speed up and cache pollution (11). Cache pollution is the degradation that happens in terms of memory bandwidth due to occupying the memory with unneeded data or wrongfully predicted data (12).

This paper is organized into eight sections. Section 2 gives the necessary background of the Markov algorithm applied in the Markov prefetchter. Section 3 describes the implementation of the algorithms used including Markov, RPT, and DCPT prefetching algorithms. Section 4 discusses the methodology of the simulation environment. Along with the configurations and workloads in our simulation. Section 5 shows the results of simulation. Section 6 discusses these results and analyse each algorithm performance. Section 7 discusses the evolution of hardware prefetching algorithms as the related work. Section 8 draws the conclusion of this paper and states the future work.

II. Background

A. Basic idea

Markov chains are a famous mathematical construction in modern probability theory (13). They describe a sequence of possible transitions between the states. Moreover, Markov chains can be easily interpreted by finite state machines (14). The process of transition between the two states should satisfy the *"Markov property"*, which states that next state must only depend on the previous state, but not on the sequence of the states, this property is also called *"memorylessness"*.

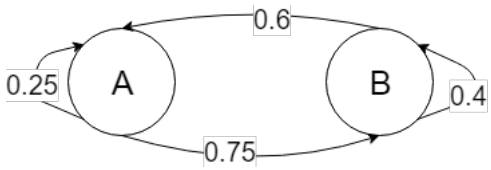

Fig. 1: Two-state Markov process

According to the diagram 1, there are 2 states with 4 possible transitions. Each number represents the probability factor of transition between the states. Let's assume, if current state of the system is A, thus it's more likely that next state will be B, since the probability for a transition from A to B is 0.75. The same applies for the B state, if our system is in the B state, the probability that next state will be A is higher than the probability that system will stay in the state B.

B. The approach

Address based prefetchers consider miss addresses as the main trigger for the prefetching process. All the states and transition probabilities are based on the stream of missed addresses inside the cache. Markov model keeps tracking the history of misses and dynamically adjusts the weights for the transitions.

In the following example it's assumed that each letter represents a memory location.

A, C, D, B, E, F, F, A, C, C, E, A, E, F, C, E

Observing the stream of these missed addresses, Markov model can be built. In this sequence, the cache tries to get access to the memory location A and gets a cache miss, and so on for the whole sequence.
In figure 2, it is clear how the weights for transitions are assigned in a Markov prefetcher model. Elaborately, the missed address C is repeated 4 times in the aforementioned sequence. Moreover, pattern "C D" occurs 50% of the time, another pattern "C F" occurs 25% of the time and, pattern "C B" occurs 25% of the time. It is clear that the Markov prefetcher will predict the next address using the recorded history of the previous missed memory locations. thus, if the prefetcher algorithm gets an address C miss, it will prefetch address D as the most probable address to be used next.

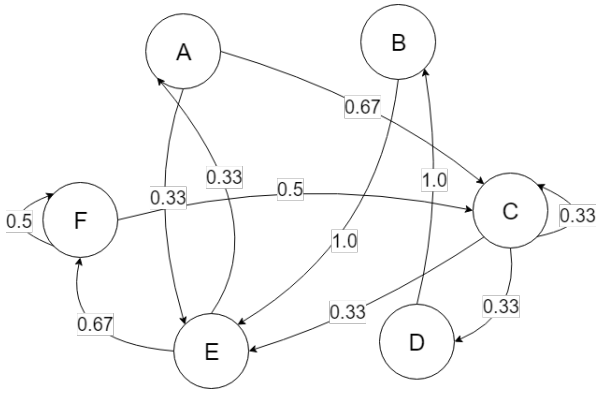

Fig. 2: Markov prefetcher model

III. Implementation

To test our implementation of the Markov prefetching algorithm, two more algorithms were implemented. RPT and DCPT. The main reason is to better analyze the Markov prefetching algorithm performance as an example of the address-based prefetching algorithms against data-based prefetching algorithms, specifically in terms of complexity, speed up and area.

A. Markov based Prefetcher

In order to predict the next address after a miss is detected, the current missed address should have been already added to the Markov chain as a *node*. Intuitively, each node should have possible *transition nodes* with certain probabilities assigned to each path. Whenever a miss occurs, this address is appended as a possible transition node to the previous missed address. After going through different implementations, and sweeping over different parameters. The following parameters, shown in table I, were chosen based on the overall accuracy and speedup achieved. The following parameter values will be discussed more thoroughly in the following sections.

Parameter	**Value**
Max Number of Nodes	6000
Max Number of Transition Nodes	4
Max number of prefetched addresses/cycle	32

TABLE I: Table of Parameter Values

However, the prefetcher implementation can't be limited by a fixed number of entries, since new addresses can be requested after a limit of 6000 is surpassed. To choose which address should be replaced by the recently missed address, a time-stamp value was added to both the nodes and the transition nodes. Thus, whenever a new address is to be added to the array of nodes or transition nodes, the least recently used (LRU) policy is used. Furthermore, if the recently missed address had already existed in the table, its time-stamp would have updated to the current time value.

Missed Address 1	Transition Node 1	.....	Transition Node 4
Missed Address 2	Transition Node 1	.....	Transition Node 4
.....	Transition Node 1	.....	Transition Node 4
Missed Address 6000	Transition Node 1	.....	Transition Node 4

TABLE II: Representation of the Markov prefetcher Table

Transitions in the Markov chain are based on the probability values. Therefore, to calculate the probabilities of transitions, the prefetcher should keep tracking the frequency of access to the transition nodes. This was implemented by adding a weight to each transition node. When it's being used/accessed, weight is increased by one. The next address to prefetch will be the transition node with the highest weight.
Since Markov prefetcher can't reference to the address if it doesn't exist as a node in the Markov chain, the overall performance can be improved by integrating a sequential prefetcher to the current algorithm. Sequential prefetching is used when missed address can't be found in the array of nodes, typically in the first cycles where the table has a few entries.

B. RPT Prefetcher

Using a table of the missed PCs, for each entry in the a table, a delta value for the stride between the current address and the next used address is recorded. Whenever there is a miss, the table is checked for the PC value, if it exists, the address calculated from the current address is added to the corresponding delta to be prefetched.

C. DCPT Prefetcher

A more elaborate algorithm was implemented for the DCPT. The challenges in DCPT implementation is the number of parameters that could be tweaked, and entirely change the performance. An example would be the size of the table, the number of deltas to be saved per PC, the number of address to be prefetched.To discuss the implementation thoroughly, we will go through each parameter.
Unlike RPT, our DCPT implementation does not start on a miss, but the algorithm checks each PC, giving it more effectiveness. Using a table, whenever a new PC arrives,

it is added to the table. The table has a maximum of 100 entries, and LRU policy is used for replacement. For each PC entry in the table, the last address accompanied with PC is recorded, as well as a queue of the calculated deltas. The delta is calculated as the difference between the last used address and the current address accompanied with the same PC, whenever it shows up again, as shown in equation 1.

$$delta_{PC_i} = Addr_{PC} - Addr_{PC_i}; \quad \forall\ PC_i = PC \tag{1}$$

where the PC_i is the current PC, and PC is the PC for a specific entry in table. For the Delta correlation, only 6 entries were used for the delta queue for each PC, also updated using the last recently used policy. The consecutive deltas are checked, if they were ever repeated in the same sequence. if a correlation is found, all the deltas in the queue are used for generating the to-be-prefetched addresses. Theses addresses are calculated by adding those deltas to the current address. The addresses are checked if they already exist in the cache, before being prefetched. Another implementation remark is that a new 32-entry buffer was added to get the latest prefetched address. This buffer is checked as well, so that the same addresses are not repeatedly prefetched over a short period of time, if they were overwritten by the cache.

IV. Methodology

SPEC CPU2000 benchmark suite (15) was used for comparing the performances of the prefetching algorithms. SPEC CPU2000 is a software benchmark product by the Standard Performance Evaluation Corp. Some of the tests that were taken into account are compute-intensive floating point performance, computational chemistry, parabolic partial differential equations, and neural network simulations.

SPEC2000 benchmark platform gives the opportunity to test the performance of the different prefetchers using the M5 cache simulator. The architecture simulator is based on the Alpha21264 super-scalar out-of-order microprocessor. It contains 2 L1 caches, a 32KB for instructions cache and a 64KB for data cache. The block size is 64B. The L1 caches do not support any prefetching. The size of the L2 cache is 1MB, with a block size of 64B. The simulated memory bus is 64 bits wide, it operates at the frequency of 400MHZ and has the latency of 30ms.

To implement the prefetcher, M5 simulator provides 3 main functions to modify - **prefetch_init, prefetch_access, prefetch_complete**. When prefetcher is executed, the first function that to be called is **prefetch_init**, all data structures are initialized inside this function. Every time when the L2 cache is accessed through the L1 cache, **prefetch_access** notifies if hit or miss was detected. After the requested block of memory is placed inside the L2 cache, **prefetch_complete** function is called.

V. Results

In this section the results that were achieved using the benchmark tests are shown. At first, some of the main parameters that will endorse the analysis are explained in the equations 2, 3 and 4, specifically the speedup, the accuracy, and the coverage.

$$speedup = \frac{execution\ time_{no\ prefetcher}}{execution\ time_{with\ prefetcher}} \tag{2}$$

$$accuracy = \frac{good\ prefetches}{total\ prefetches} \tag{3}$$

$$coverage = \frac{good\ prefetches}{cache\ misses\ without\ prefetching} \tag{4}$$

Additionally, one more important parameter was taken into consideration, when assessing each algorithm, which is the sizes of the table used for each algorithm, those can be observed in Table III.

Algorithm	Size
Markov	6000 x 4
DCPT	100 x 6
RPT	100 x 1

TABLE III: Table of sizes

To justify the numbers shown for the table sizes, figures 3, 4, are used to show the speedup and accuracy achieved upon using different table sizes for the Markov implementation respectively. This will also be used in the next chapter for better understanding the points of strength and weaknesses for each algorithm. Finally, figures 5, 6 and 7 are showing the different parameters that were taken into account as the speedup, accuracy of the prefetched data, and the coverage for the different implemented algorithms for all the tests.

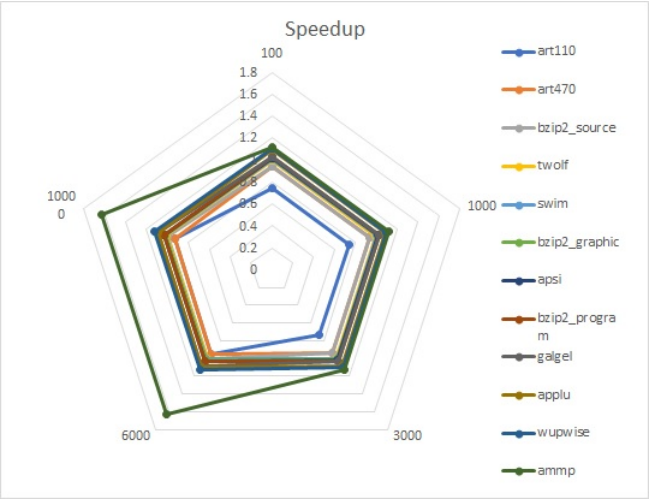

Fig. 3: Speedup over 5 table sizes 100, 1000, 3000, 6000, 10000 for Markov Algorithm

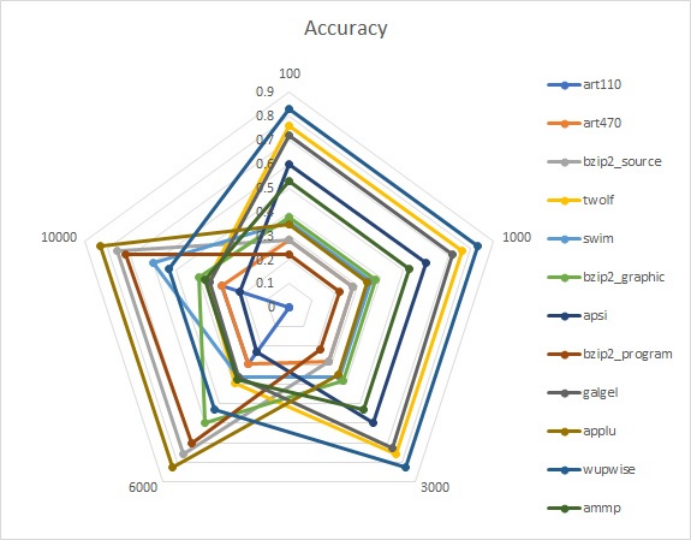

Fig. 4: Accuracy over 5 table sizes 100, 1000, 3000, 6000, 10000 for Markov Algorithm

VI. Performance Analysis

In the figure 5, the different speedups are described that were obtained when using the different algorithms. In terms of Speedup, most of the algorithms got close results in different tests. In “ammp”, “wupwise” tests, we notice a slight difference between the algorithms. For Markov and DCPT, they achieved higher results than RPT, especially in the “ammp” test. When analyzing the “ammp” test pattern, the addresses are repeated in wide intervals approximately every 5000 addresses. This leads to a poor performance for RPT which only uses a 100 entry table. As DCPT for instance uses deltas, that enabled the algorithm to get higher result even when compared with 6000 entry Markov. In figure 3, the reason to choose a high number (6000) entries for the Markov table is explained, as it is clear that for the “ammp” test, 6000 entries at least were needed to achieve a speedup of about 1.6. When comparing this with figure 5, DCPT still exceeds the speedup achieved with 1.8 in the “ammp” test. In overall, the DCPT achieved a higher speedup of about 1.10, while Markov got 1.06, and RPT got 1.02.

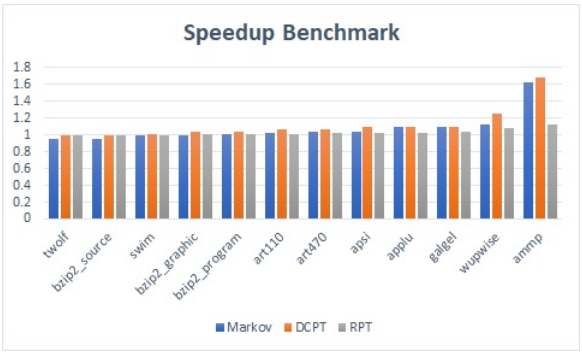

Fig. 5: Speedup Benchmark results for the implemented algorithms

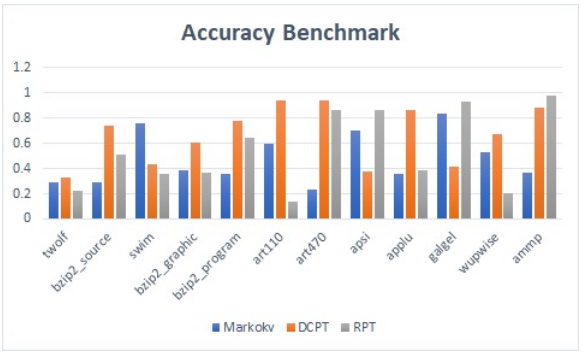

Fig. 6: Accuracy Benchmark results for the implemented algorithms

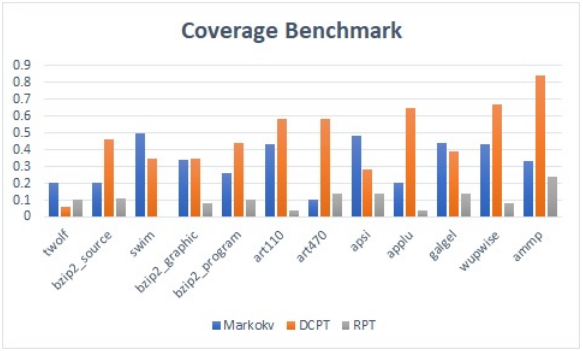

Fig. 7: Coverage Benchmark results for the implemented algorithms

In terms of Coverage, DCPT performed much better than other algorithms. That gives us an indication that the DCPT algorithm has a high coverage through using deltas, which gives the correlation the higher accuracy. Also, having an additional buffer in our implementation that prevented prefetching data that was prefetched recently, and might have not been useful. Also, as mentioned before, the RPT only prefetches one address, while DCPT prefetches maximum 6 addresses. On the other hand Markov has to prefetch around 32 address every cycle to achieve a speed up compared with DCPT. That again implies that the Markov algorithm suffered from the fact that it is address based.

Finally, the accuracy as understood from the coverage results Markov algorithm had worse results than DCPT and even RPT in some of the tests. In “ammp”, Markov got a very low accuracy pattern, that is from the fact that Markov prefetches 32 addresses per cycle. This can be investigated better in figure 4. In this figure, it is shown that there is a trade-off between some of the tests over the other, specifically using large tables favored “applu”, “bzip2_program”, and “bzip2_source”. The accuracy results shed light on one of the main differences in Markov coming from the fact that it is address based, to achieve high speedup, it needs to prefetch a lot of addresses, as the probability only is not a good indicator to filter some of these candidates.

One last point worth mentioning, is the size of the tables used. The tables for the basic RPT were small, and achieved worst results as expected. On the other hand Markov used 60 times

the window size and got worse results when compared with DCPT, giving a good implemented DCPT the upper hand over Markov and RPT algorithms.

VII. Related Work

The main focus of this paper was hardware prefetchers, that among other reasons is because of the fact that hardware prefetchers are more performance friendly, in the sense that they do not share the execution bandwidth with the instruction cycles running at the same time (5). For hardware prefetcher implementation, several mechanisms and techniques were employed. The first idea came from recognizing the memory access pattern, thus predicting the next probable address to be fetched, and load it to the cache. The basic pattern was just the sequential prefetcher (16), where only the sequential address will be prefetched after a cache miss. Stride direct prefetching (SDP) (17) came next, it worked on the same concept of spatial locality, instead of deducing the next address from the cache misses, it recorded the address being fetched by the processor every time the program counter is incremented. Based on the difference of the sequentially addressed places in memory, the next address to prefetch is calculated.

The disadvantage of using the sequential and SDP techniques is the cache pollution due to the simplicity of the pattern being predicted as not programs access memory sequentially. A next step for hardware prefetchers started, when reference prediction tables (RPT) (10) were first introduced. More complex patterns were captured using the RPT. By using the current Program Counter (PC), and the relative reference address. when the same PC is repeated, the new reference address is saved, and a delta is calculated as the difference between the old and the new reference addresses. This is called the training state. Then the prefetching state is triggered, when a delta matches with the previously calculated one.

In the RPT, the idea of having a table with recorded values of memory access was first initiated. As a follow work, Program Counter/Delta Correlation Prefetching (PC/DC) (9), Delta Correlation Prediction Table (DCPT) (18) and Markov prefetchers (11) all centered upon having a history table and trying to recognize the pattern using the cache misses history. At this point the drawback of using one algorithm rather than the other is the memory bandwidth pollution, the size of the table needed, and the incline in the cache hits leading to the speed up achieved. From (5; 19), it seems that DCPT has an upper edge from other algorithms in terms of speed up.

Starting from the idea of adaptive prefetching (20), a new class of prefetchers were introduced. the adaptive algorithms used a performance metric measurement to adapt, either by being more aggressive if it was succeeding, or the other way around if it was failing. At first, this class of prefetchers did not achieve higher performance than the non-adaptive methods. This mainly comes from the fact that the measure of either the algorithm is being successful or was not highly accurate, leading to an overhead in the error in the prefetcher performance.

In the last few years, several algorithm targeting hardware prefetchers were proposed to tackle this problem. In (6; 19; 21; 22; 23; 24) machine learning algorithms were used within the hardware prefetcher architecture. In (19), a two-level prefetcher were proposed that exceeded the conventional prefetchers in terms of having less cache pollution. The machine learning that was used was only based on a one layer of preceptrons, the training set was the global history table introduced by the means of the first layer prefetcher, a Markov prefetcher. The perceptron layer helped filtering some of the decisions taken by the Markov prefetcher and that improved the overall performance. In (21), a program characterization and machine learning were used to choose a certain hardware prefetcher to match a certain application. In (22), 225 prefetching configurations were used to highly customize for the different applications running on the IBM POWER8 processor.

In (6), the problem of having non-spatio-temporal memory access was addressed. The paper introduced the concept of semantic locality, which uses inherent program semantics to characterize access relations. With reinforcement learning, they were able to detect the semantics locality patterns in memory access. At this point the machine learning algorithms were being used as a supporting part in the hardware prefetcher either to filter decisions or to highly customize for a certain application to ensure that the prefetcher is being adaptive in the right manner. In the future, more machine learning algorithms will be integrated in the implementation of hardware prefetchers, especially the algorithms that learn patterns from relatable series of training date as long short term memory(LSTM) and recurrent neural network (RNN). In 2018, a thesis was published on a similar idea (24), where LSTM was used as a hardware prefetcher.

VIII. Conclusion

In this paper, we compared Markov prefetcher as an address based prefetching scheme against common data-based prefetching schemes as RPT and DCPT algorithms. We also investigated the specific details that made each algorithm better in different tests. By adjusting the window sizes of these algorithms, tests results were analyzed and, consequently, sizes of the tables were chosen according to the most optimal performance of each algorithm.

When analyzing the overall results, it is obvious that the DCPT performance was the best. However, Markov prefetcher can be developed further by transforming it into distance based prefetcher. This improvement is achieved by storing deltas of two consecutive addresses instead of addresses themselves. That will allow us to decrease the table size and make this algorithm more flexible, since one delta can give more information to the prefetcher than a single address. Hybrid Markov-Distance prefetched should be also investigated. In this implementation addresses are still stored as nodes, but transition nodes are replaced by deltas.

Overall, depending on the specific test, each algorithm has its own advantages. Thus, it is clear that context of the application should be taken into consideration and the prefetcher choice will rely on this context. This applies not only to the type of applications it will operate on, but also to the hardware environment.

REFERENCES

[1] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, vol. 23, no. 10, pp. 20–24.

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38.

[3] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. G. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, pp. 6–20, 2014.

[4] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 152–162, 2011.

[5] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 35:1–35:35, Aug. 2016. [Online]. Available: http://doi.acm.org/10.1145/2907071

[6] D. Guttman, M. T. Kandemir, M. Arunachalam, and R. Khanna, "Machine learning techniques for improved data prefetching," *5th International Conference on Energy Aware Computing Systems Applications*, pp. 1–4, 2015.

[7] J. Lee, H. Kim, and R. W. Vuduc, "When prefetching works, when it doesn't, and why," *TACO*, vol. 9, pp. 2:1–2:29, 2012.

[8] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *ICS*, 2004.

[9] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: an adaptive data cache prefetcher," *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pp. 135–145, 2004.

[10] Y. P. Chen, H. Zhu, and X.-H. Sun, "An adaptive data prefetcher for high-performance processors," *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 155–164, 2010.

[11] D. Joseph, "Prefetching using markov predictors," *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, pp. 252–263, 1997.

[12] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 63–74, 2007.

[13] L. R. Rabiner and B. H. Juang, "An introduction to hidden markov models," *IEEE ASSP Magazine*, vol. 3, pp. 4–16, 1986.

[14] E. Seneta, "Markov and the birth of chain dependence theory," *International Statistical Review*, vol. 64, 12 1996.

[15] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *IEEE Computer*, vol. 33, pp. 28–35, 2000.

[16] K. M. Kavi, "Cache memories," 1998.

[17] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *MICRO*, 1992.

[18] M. Grannæs, "Reducing memory latency by improving resource utilization," 2010.

[19] H. Wang and Z. Luo, "Data cache prefetching with perceptron learning," *CoRR*, vol. abs/1712.00905, 2017.

[20] F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," *1993 International Conference on Parallel Processing - ICPP'93*, vol. 1, pp. 56–63, 1993.

[21] S. Rahman, M. Burtscher, Z. Zong, and A. Qasem, "Maximizing hardware prefetch effectiveness with machine learning," *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 383–389, 2015.

[22] L. Peled, S. Mannor, U. C. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 285–297, 2015.

[23] M. Li, G. Chen, Q. Wang, Y. Lin, H. P. Hofstee, P. Stenström, and D. Zhou, "Pater: A hardware prefetching automatic tuner on ibm power8 processor," *IEEE Computer Architecture Letters*, vol. 15, pp. 37–40, 2016.

[24] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher: A case study," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, 2017.