Proposal to NEC:

Adaptive Prefetching using Neural Networks

John Cavazos and Darko Stefanovic

March 27, 1997

1 Introduction

The performance gap between the processor and its caches has been increasing over recent years. An even more prominent gap is found between the processor and main memory. Because of these performance gaps, a hindrance like a cache miss can be a serious detriment in high-end processors. Cache miss ratios can be reduced by smart replacement policies or by prefetching. We concentrate on prefetching here, because it holds the potential to prevent cache misses in the first place. In order for any mechanism to be effective in prefetching, it must be able to discriminate patterns (e.g., arithmetic progressions or cyclic recurrences) in past memory accesses in order to predict future memory accesses. Simple techniques have been proposed and shown to achieve limited success. The task is also amenable to neural networks. Furthermore, neural networks can, in principle, detect complicated patterns that other methods cannot. These complicated patterns are important because they arise quite often in programs with pointer-intensive computations (e.g., object-oriented programs) or sparse matrix computations. It is a matter for experimentation to determine the effectiveness of a neural network in detecting these complicated patterns. In section 2, we discuss why neural networks will be effective in prefetching. In section 3, we discuss the version of the problem we will tackle and the auxiliary structures that make this version more amenable to neural networks. In section 4, we elaborate on the experimental infrastructure to be used.

2 Background

Memory accesses in most programs exhibit some form of regularity. Simple hardware and most simple prefetch mechanisms (such as first order Markovian models [5]) tend to do well only when the regularity is in the form of temporal and spatial locality, or when the regularity is in the form of simple arithmetic progressions such as those found in loop nests in some scientific codes. More complicated regularity found in programs that have pointer-intensive computations (e.g., object-oriented programs) or sparse matrix computations will present problems to many proposed prefetch mechanisms[1]. It is our hypothesis that neural networks, on the other hand, will be considerably more effective at prefetching for these languages—to the extent that the implementation complexity and cost of neural network schemes can be justified on performance grounds for object-oriented languages.

In order for prefetching to be effective, any prediction technique must satisfy the following primary criterion; if possible, the prediction technique should satisfy the secondary criterion as well.

Primary: It must have a very *high correct prediction rate*. If it hasn't, then the added congestion on the bus and the replacement of possibly "live" cache blocks by incorrect data will outweigh the benefits of any correct prefetch predictions.

Secondary: It must provide an accurate *confidence level* for the predictions it makes. This permits the prefetching mechanism to forgo prefetching in cases of low confidence and thus avoid the congestion problems.

Several prefetching schemes satisfy the primary criterion, but only to a limited extent. They are limited either in the patterns they recognize or in the number of predictions they can make. No scheme that we know of satisfies the secondary criterion. An appealing aspect of using neural networks for prefetching, as discussed in the next section, is that they are able to satisfy both criteria.

3 The Problem

Intelligent prefetching is beneficial in several areas of computer design. At most levels of a computer's memory hierarchy, intelligent prefetching is desirable be-

[1] Some of these accesses can be handled by a recently proposed prefetch mechanism. The method proposed by Mehrotra and Harrison [4] is able to predict and prefetch one memory reference in advance, as opposed to our method which will be able to predict several memory references in advance.

cause it is effective in hiding latencies in lower levels of the hierarchy. Intelligent prefetching is also desirable in multimedia systems, databases, and distributed systems. The version of the prefetching problem that we will concentrate on is prefetching at the highest level of a computer's memory hierarchy, i.e., the first level data cache.[2]

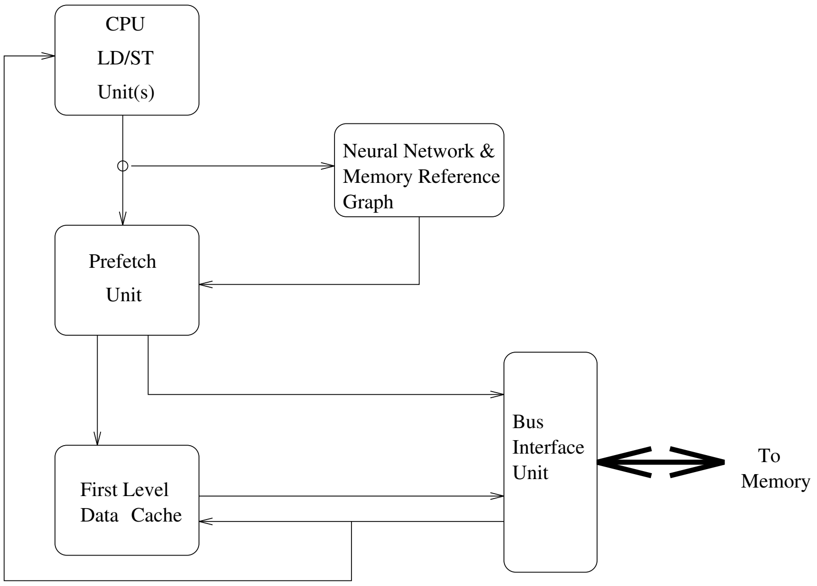
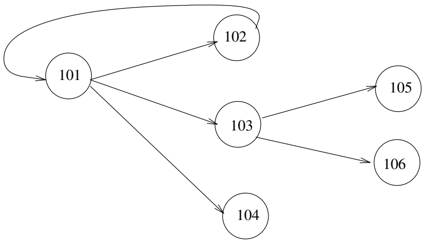


Figure 1: Envisaged CPU with intelligent prefetcher.

In Figure 1, we sketch how our intelligent prefetcher will fit into a modern CPU. Most modern processors already contain a prefetch unit, usually implemented using simple schemes to control what they prefetch. We propose adding an additional component which will make more intelligent retrieval choices for the existing prefetch unit. The prefetcher is located between the load/store functional units and the first level data cache. The new component is located between the functional units and the prefetcher. As memory references are issued by the functional units, the requested addresses are stored in a table. An example is given in Figure 2. Associated with each address in the table is a list of indices: *an adjacency list*. Whenever an access to location a is immediately followed by an access to location b, we place the low order bits of b into an available entry in the adjacency list

[2] Intelligent prefetching for instruction caches requires intelligent branch prediction. Incidentally, branch prediction using neural networks has been researched [2], and results are quite promising.

for a. The table, in effect, is a representation of a graph of memory references of the program (the *memory reference graph*[3]), where each row of the table contains a vertex (i.e., the address of the memory referenced) and an adjacency list for that vertex.



		Adjacency list elements: 0	1	2	3	4
	● ● ●					
101	...1100101	102	103	104		
102	...1100110	101				
103	...1100111	105	106			
104	...1101000					
105	...1101001					
106	...1101010					
	● ● ●					

Figure 2: An example of a memory reference graph and the table representing it. The saturating counter and recent access bits for each adjacency list element have been omitted.

The table is likely to be very large.[4] The table size needs to be large enough to accommodate substantially more entries than the first level data cache, otherwise all the data represented in our table would already be in the first level data cache, and prefetching would be useless. The table will be accessed similarly to a cache (i.e., indexed by the lower portion of the address), and will have a small degree of associativity.

Instead of requiring a neural network to learn over and predict sequences of 32- or 64-bit numbers (the raw memory addresses), we will train with indices into the table.[5] The table simplifies the information used to train the neural network and also simplifies what it is asked to predict. The output nodes of the neural network correspond to outgoing edges in the memory reference graph from the current vertex. Each output node corresponds to an element in the adjacency list. The output node with the value closest to 1 corresponds to the edge which the neural network predicts will be traversed in the memory reference graph. The

[3]The memory reference graph here is similar to the access graph in [1].

[4]The table size and its associativity will be variables for experimentation.

[5]Another technique that has the potential of simplifying the learning task even more is training the neural network with positions in an LRU stack of memory references [7].

confidence of the prediction is determined by the number of possible edges and the relative closeness to 1 of the alternative, non-predicted edges. We can allow the neural network to predict further references by feeding its predictions onto itself.[6]

Having a table of fixed size to store memory references has some problems, [7] which require three additional hardware components to solve.

- For each element of the adjacency list, we will require an n-bit[8] *saturating counter* corresponding to the number of traversals down each edge. Once the adjacency list is full, these counters will allow us to select those elements of the list corresponding to edges that have been traversed few times, in order to overwrite them and make room for new edges.

- In addition, we will keep a *recent access bit* which will allow us to find an element in the list to evict if all elements have a saturated counter (that is, if all counters have reached their upper limit). Even using this method, the neural network may occasionally make predictions that are based on the training that partly happened before a list entry was overwritten, so the traversals of the edge that previously occupied that list entry interfere with the current one. We believe that evicting the least recently used edges is a good tie-breaking policy when all edges are frequently traversed.

- Adjacency list entries in one row point to other rows, however the identity of a row can change between the time the adjacency list entry is created and the time it is consulted. This happens when the old address stored in the row is evicted to make room for a new address. The resulting aliasing would give rise to the incorrect prediction of the new address, rather than the old address which has been lost. Instead, we will compute an m-bit summary of the old address and store it along with the adjacency list entry. When an edge is predicted, we can check if the stored summary equals the summary computed for the correct address in the predicted row. A few bits for checking cannot completely alleviate conflicts due to aliasing, but they can greatly reduce the number of incorrect predictions sent to the prefetch unit.

[6] This will require two things: (1) flipping a switch telling the network to stop learning and (2) buffering the true tapped delay line of the previous memory references (see below).

[7] Capacity and aliasing problems arise, but these are common in computer hardware and generally not insurmountable.

[8] The correct n is another variable for experimentation. We believe that a small number of bits will suffice.

As an alternative to storing indices in the adjacency list elements, one could store the actual raw memory addresses. With a few bytes for each adjacency list element (and a few bits for the m-bit summary), the space required, though, is still less than the 64 bits required to store a memory address.

We will begin our investigation using time delay neural networks, Figure 3. In a recent study, time delay neural networks were shown to be adequate in on-line recognition of temporal patterns and capable of converging to correct solutions quickly [6]. Subsequently, other neural networks, e.g, temporal recurrent neural networks and self-organizing neural networks [3], will be explored. The inputs to the neural network are indices into our memory reference table.

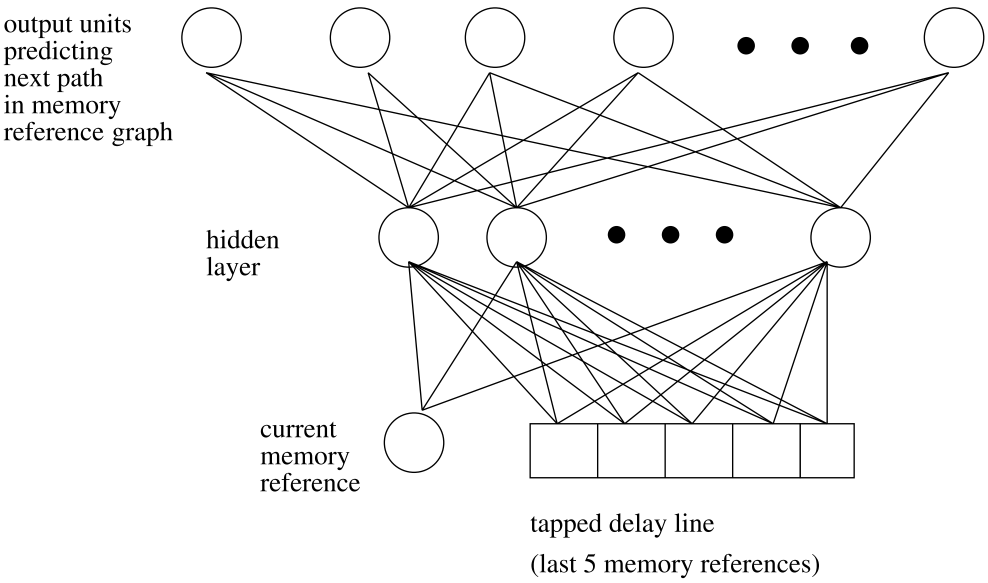


Figure 3: A Time Delay Neural Network.

The technique described above will allow the neural network to predict cyclic recurrences from past memory accesses, if they exist. For predicting new memory references that may occur, i.e., predicting the next reference if the last several references formed an arithmetic progression, we will train another neural network on a small number of the least significant bits of previous memory addresses and allow it to predict the next low order bit sequence to appear in the arithmetic progression. The prefetch unit will be allowed to choose the prediction with the highest confidence level of both neural networks.

4 Experimental Infrastructure

Initially, we will train and test our neural network over a trace of memory references. The neural network will learn on-line, predicting future memory references from past references it has seen. Many such memory reference traces have been generated for our previous work on memory management [8]. These traces are mostly of Smalltalk and SML/NJ programs. It is also desirable to generate memory reference traces for the Spec95 benchmark suite, which we shall do using the instrumentation tool ATOM, from Digital. We also have several benchmarks written in compiled object-oriented languages (C++ and Modula-3) that will similarly be instrumented to output memory references. Benchmarks written in object-oriented languages are particularly interesting because pointer chasing, due to the indirect function calls (i.e., method invocations), occurs more frequently in these languages. For approximating actual run time performance benefits using our prefetching technique, we will build a cache simulator using ATOM. In this cache simulator we will embed our neural networks. This will allow a simulated prefetch unit to receive hints from the neural network and place the suggested cache blocks into the simulated cache. Our experiments will also include varying various cache parameters, such as total size, block size, and replacement policy, in order to examine the relative benefits of prefetching in interaction with different cache structures.

References

[1] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50:244–258, 1995.

[2] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. In *Conf. on Prog. Language Design and Implementation (PLDI '95)*, pages 79–92, 1995.

[3] Daniel L. James and Risto Miikkulainen. SARDNET: A self-organizing feature map for sequences. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 577–584. The MIT Press, 1995.

[4] Sharad Mehrotra and Luddy Harrison. Quantifying the performance potential of a data prefetch mechanism for pointer-intensive and numeric programs.

Technical Report 1458, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, November 1995.

[5] Vidyadhar Phalke and Bhaskarpillai Gopinath. A miss history based architecture for cache prefetching. In Henry G. Baker, editor, *Second International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science, Kinross, Scotland, September 1995. Springer-Verlag.

[6] M. F. Sakr, C. L. Giles, S. P. Levitan, B. G. Horne, M. Maggini, and D. M. Chiarulli. On-line prediction of multiprocessor memory access patterns. In *IEEE International Conference on Neural Networks*, pages 1564–1569, 1996.

[7] Jeffrey R. Spirn. Distance string models for program behavior. *IEEE Computer*, 9(11):14–20, November 1976.

[8] Darko Stefanović. Object-level tracing toolkit: design, implementation, and purpose. September 1995.