Paging Behavior of Knowledge Networks

Roberto Bisiani
Department of Computer Science
Carnegie-Mellon University, Pittsburgh, Pa. 15213

August 10, 1977

Abstract

The problem of efficiently storing and retrieving a special class of knowledge data bases, namely state space networks, is tackled. This problem has many similarities with the storage and retrieval of program code and data in virtual memory systems, therefore special attention is paid to the already existing techniques for virtual memory systems. Differences and similarities are examined and a few techniques for storing and retrieving networks are presented. All the proposed "software" strategies fail to attain a very fast retrieval, as desired in real time applications, therefore a solution using a bubble memory system is examined. Finally a few experimental results, obtained both with simulation and with measurement of real systems, are presented.

1 Introduction

Several problem areas in artificial intelligence (AI) require the use of large knowledge data bases. Typical of such tasks are speech, vision, natural language, and so on. One of the commonly used representational structures is the network, that is an oriented graph. In this graph, depending on the application, a certain amount of data is associated with the nodes, often called states, and the arcs. The retrieval of the information stored in the network is directed by a procedure called "search" that follows certain "paths" (sequences of adjacent and distinct arcs) according to particular rules. The search process terminates when either a certain state is reached or a certain value of a given function, that defines the problem, is obtained. There are many ways in which a program can search a network; we are interested in a search where:

the searching strategy does not fully define the path;

many different paths can be active at the same time.

Moreover we will consider the network as a read-only data base. That is, we will not allow any dynamic modification of the graph structure or of the data associated with nodes and arcs. This restriction is consistent with the behavior of many AI programs if we can store the data generated during the search in a separate structure. This is a reasonable assumption since only the dynamic information relative to a limited number of nodes and arcs must usually be known at a certain time.

The size of networks usually ranges from many thousands to millions of states (precise numbers for a speech understanding system will be given in **3**). This obviously implies the use of cheap secondary storage media. Moreover the number of states accessed for the solution of a single problem is usually very high: it is not uncommon to access 50% or more of the total number of states.

We identify two main situations which require high performance storage and retrieval techniques of AI networks:

> the network is very big and therefore the number of states searched is so high that a considerable amount of time is spent retrieving information from the secondary storage;
>
> the solution of the problem must be found in a bounded time (e.g. real time speech recognition).

In order to reduce the search time we can either use specialized storage and retrieval techniques (that minimize the number of secondary memory accesses), or rely on some specialized hardware (that minimizes the time required for a secondary memory access).

We will first take into consideration the commonly used storage and retrieval techniques; at first sight there is a strong similarity between the problem of storing and retrieving programs and data in virtual memory systems and the problem of storing and retrieving networks. Unfortunately this turns out to be false for two main reasons:

> the behavior of a common program, in terms of sequences of references, is usually different from the behavior of the searching process of an AI network;
>
> the performance figures, in terms of number of secondary memory accesses, needed for a good behavior of programs are less limiting than the ones needed for AI tasks.

In the following chapters we will first discuss the commonly used storing and retrieving techniques and explain to what extent they can be applied to the storage and retrieval of networks. Then we will briefly analyze, as a case study, the behavior of a speech recognition system developed at Carnegie-Mellon University. This system will be used in the remaining of the paper to evaluate a few different storing and retrieving techniques. The experimental data that will be presented have been obtained both with simulation and with measurement of real systems.

2 Commonly Used Techniques

We are interested in the following techniques:

clustering the information into secondary memory blocks (pages);

managing the main memory.

2.1 Clustering procedures

In paged virtual memory systems the program code must be partitioned into fixed length pages so that it can be stored efficiently on the block oriented mass storage media currently available. The layout of the program in its virtual space, generally decided by the linker, is influenced by the order in which the programmer inputs the instruction blocks and declares the data blocks. Since the early applications of virtual memory systems it was recognized [Comeau, 67] that the performance, obtained by simply committing to the linker or to the compiler the task of partitioning the program, could be improved with a careful reordering of the code into the pages. Manual rearrangements made by experienced programmers gave good results but, especially as far as complex programs are concerned, computer aided rearrangements gave better results [Ferrari, 76].

Since those early studies, many important results have been achieved in the field of automatic program restructuring. Three different kinds of procedures have been investigated:

static procedures that base their decisions on purely static information about the program, that is, on its text. These procedures can be applied at compiling or loading time [Ramamoorthy, 66; Lowe, 70; Ver Hoef, 71; Baier, 72];

dynamic a priori procedures that utilize information about the behavior of the program gathered during one, or a few, of its runs [Hatfield and Gerald, 71; Ferrari, 74];

dynamic a posteriori procedures that partially rearrange the program during every run using information gathered during the run itself [Baier and Sager, 76].

The currently used procedures are the dynamic a priori ones. The static procedures have been outperformed by the dynamic a priori ones and the dynamic a posteriori ones seem to be still beyond the state of the art. Both static and dynamic a priori procedures perform the restructuring in three steps:

first the code is devided into "atomic" blocks smaller than the secondary memory blocks (blocks);

then the cost of not grouping together two blocks in one page is computed;

finally the blocks are clustered into secondary memory pages by an algorithm that minimizes the inter-page cost.

The crucial step is obviously the computation of the cost; the kind of information it uses and the phase in which this information is collected fully characterize the restructuring procedures.

There are a few difficulties in using the outlined procedures for AI networks:

there is no straightforward way of partitioning the net in atomic blocks in order to cluster them in pages. Program references are mostly sequential if we observe them through a small time window (10-100 references). It is obviously correct to cluster the sequential parts of a program into blocks. The network structure does not suggest any pre-clustering: it is like a program containing only branches.

the cost-computing procedures are based on program structure, completely new cost computing algorithms must be found in order to use them for AI networks;

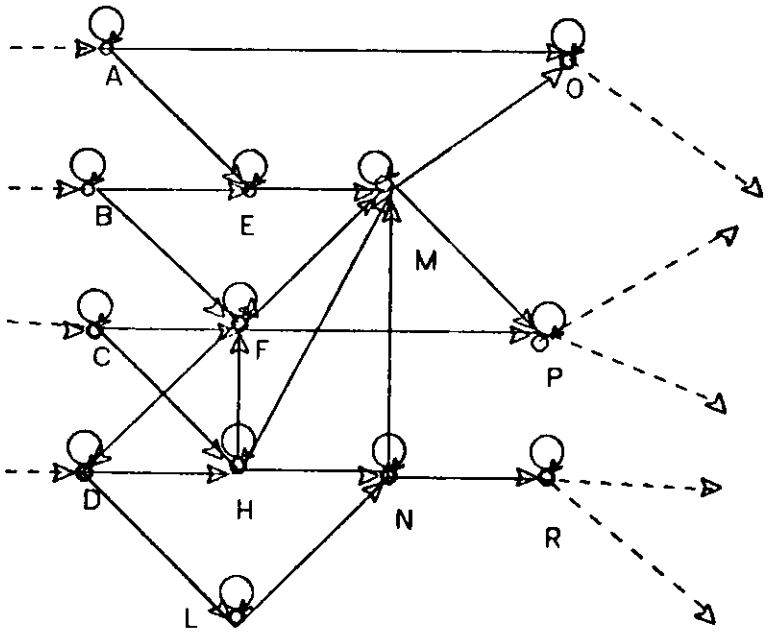
the dynamic procedures can be used only if the reference behavior is mostly data independent. While this is true for many, but not all, programs, the search process is strongly data dependent (under the assumptions made in 1).

In Fig. 1 we show part of a network and some of the possible state reference strings. Suppose to keep a list of "active states" the system uses the following search process:

all the sons of the states in the current active list are examined at the same time;

a new active list is generated selecting a subset of examined sons.

The reference string is simply a sequence of active lists (active list at time t1, active list at time t2,...). Every state in the active list at a certain time locates a possible path in the network. The behavior of the search of a given single path is reasonably predictable, e.g. while state A will never be directly followed (in the reference string) by state M, states O and P will probably be accessed after state M. Unfortunately many paths are active at the same time and there is no fixed relation between the behavior of different paths. Every subset of the states in the network can be a valid "active list" and the sequence of active lists during a certain search is critically dependent on the input data. Therefore we cannot apply any behavioral consideration based on data collected in a certain run to other runs. A dynamic a priori restructuring procedure based on data collected during the run modeled by string 1 would probably cluster M and P in the same page (they are referenced one after the other in string 1). This choice would be wrong in the case of string 2.



string 1: . . . | ABD | EHL | MFN | MR | OPR | . . .
string 2: . . . | ABD | EHL | MF | OFD | OML | . . .
string 3: . . . | ABD | ABF | EMH | EMOPF | . . .
string 4: . . . | ABD | ED | MHL | MFN | MDR | . . .
| t1 | t2 | t3 | t4 | t5 |

Fig.1 Part of a Network and Possible Reference Strings

2.2 Managing the main memory

Most of the efforts made to improve the performance of memory hierarchies have been spent in devising main memory managing strategies that could minimize the number of secondary memory accesses. These strategies can be divided into:

fetching policies that decide when a given page will be transferred into the main memory and

replacement policies that decide where in the main memory a given page will be transferred (that is what page currently in memory it will replace).

In the case of AI networks we have the following restrictions:

the main memory has a fixed size,

the network is a read only data base, therefore no write-back to secondary memory of a page selected for replacement is necessary.

Fetching policies can be divided into demand policies and preloading policies. Demand policies simply load a page when it is needed, preloading policies try to load a page before it is needed.

Block preloading policies cannot reduce the number of secondary memory accesses but only reduce the number of program interruptions due to secondary memory accesses. Therefore they can be useful in the management of multiprogrammed systems but have little influence in the case of AI networks under our assumptions. It should also be noted that preloading can be useful when the tentative preload can be overlapped by the search process. This requires that part of the availble main memory be devoted to the tentative loading, while the other part still contains the pages currently used by the search. Therefore there is a trade-off between the speed improvement attainable with preloading and the higher number of faults due to the smaller working memory. Moreover the preloading strategies now available [Joseph, 70; Madnick, 73; Joseph, 70] only have a limited accuracy.

It is an open question if very accurate preloading strategies can be devised in the case of problem oriented management policies, where the future behavior of the reference string is also dependent on some external real-time signal.

Preloading strategies will not be considered any further in this study since the main goal is to lower the number of secondary memory accesses. We will therefore always refer to demand policies in the following.

Replacement policies have been widely investigated because they deeply influence performance. These policies usually exploit the locality of programs, that is the tendency of making the same reference again in a short time. Unlike restructuring policies, replacement policies are executed during the run of the program and therefore, at least with today's computer architectures, their computational needs must be bounded. Moreover these policies use information gathered during the run: collecting this information usually requires specialized hardware. Thus the policies commonly investigated and used are limited to the very simple ones.

The possibility of using the classical replacement policies in the case of AI networks is therefore limited by the amount of locality that the search process presents. The experimental data presented in 3 show that, in terms of locality and in the specific case, the search process references resemble common programs references. This means that the classical main memory management policies can still be efficiently used. Of course the knowledge of the particular process and the availability of a specialized computer architecture can encourage the development of specialized strategies that use some internal data of the search program and require a more complex computation than the classical strategies.

3 A Case Study

3.1 Description of the task

The practical problem that triggered this study on AI nets restructuring was the need of running Harpy [Lowerre, 76], a connected speech understanding system developed at Carnegie-Mellon University, requiring very large networks. Harpy uses a network to represent all the syntatic, lexical, and phonetic knowledge needed for recognition. The live input is sampled, preprocessed and segmented in such a way that about every 40 ms the search process receives a full set of input data representing the last 40 ms of speech. Harpy's word accuracy is now about 98%. Harpy's speed, on a general purpose single processor (Decsystem 10, KL-10) is, for many non-trivial tasks, just a few times real time and could probably reach real time if a specialized processor were used.

Current networks sizes are a few thousand states and a few ten thousand arcs; every state uses about 45 bits of local information plus a variable number of bits to identify the successors of the state (e.g. a state with three successors needs 93 (45 +48) bits of main memory). The number of states and the complexity (in terms of interconnections) of the network is expected to grow when more and more complex grammars and larger dictionaries are used.

Unless otherwise noted, the data presented in the following sections is averaged over 30 typical utterances. Two different networks were used, both representing a 256 words dictionary; one (AIS05) has a low branching factor (average number of successors of states) and the other (AISF) a high branching factor. A high branching factor usually implies a very complex syntax. The biggest network used so far with HARPY is a low branching factor network using a 1011 words dictionary. Searching this network requires fewer secondary memory accesses than required by a high branching factor network representing a 256 words dictionary (AISF). The branching factor of the network, rather than the dictionary size, seems to be the driving figure in characterizing the complexity of a given Harpy task. In fact, searching AISF is the most demanding task from the point of paging and therefore, no results are presented for the 1011 network.

The input data for the various runs was not live data but either a file containing the preprocessed input data (for the two implementations) or a non-reduced trace of the search process (for the simulations). This assures the congruence between the data collected in different experiments.

3.2 Empirical behavior analysis

Some data has been collected to analyze the behavior of the Harpy search (Fig.2).

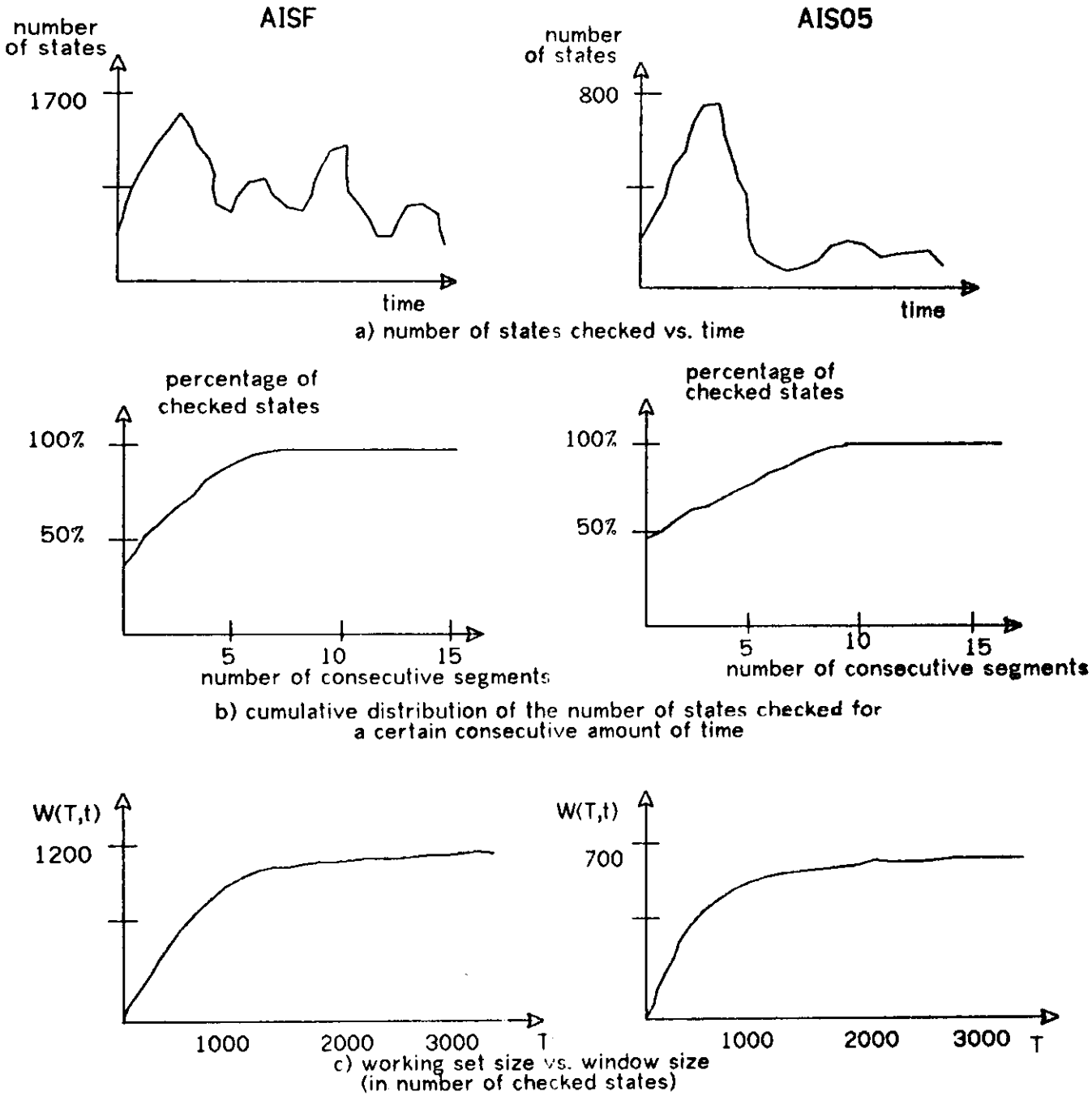


Fig.2 Behavior of the Harpy Search

Number of states searched. Fig. 2a shows the number of states searched vs. time during a utterance. Typical of Harpy search are the high peaks that represent a relative "confusion" of the search process. These peaks are responsible for most of the secondary memory accesses.

Cumulative distribution of the number of states that are checked for a certain amount of consecutive time (Fig.2b). The curve shows that 50% of the states are checked for just one or two consecutive segments.

Average working set size (in number of checked states) (Fig.2c). The average working set size is a good measure of the average locality of references. Typical working set size vs. window size curves have been published for both program and data references [Denning, 68; Rodriguez-Rosell, 76]. The working set of programs shows a rapid decrease in slope beyond a window size value while the the working set size of data references is almost a linear function of the window. The Harpy search shows, in terms of locality, a behavior very similar to the behavior of programs; this allows the use of classical replacement policies.

4 Classical Techniques: Static Reordering

A static reordering of the network has been attempted and the Harpy search has been simulated with this network in order to find the average number of secondary memory accesses in one second of speech (Fig. 4).

As described in **Section 2** the key problem for static reordering is the definition of the cost of not clustering together two blocks of information. We chose the state as the unit information element and defined the cost of not clustering together two arbitrary states A and B (where B is a successor of A) as

$$\frac{1}{(\text{number of successors of A})^2}$$

(The cost of clustering together two states, that are not connected by an arc, is considered to be infinite). Roughly, this means that only the states with a small number of successors will be clustered together with their successors. This is consistent with the tendency of the search to follow just a few of the successors of a high branching factor state. In order to provide some idea of the effectiveness of the proposed cost function, a random reordering was produced and used in the experiments under the same conditions of the other reordering. Both reordering policies have been applied with different frame sizes. AISF was run only with 1K frames since, because of its high branching factor, some of its states do not fit into a single page, smaller than 1K, with their successors. (This constraint was requested by the physical storage scheme used and has been released for the experiments presented in **Section 4**).

Different memory sizes have been simulated, including an infinite memory. The replacement policy used was a slightly modified LRU. The algorithm implementing this

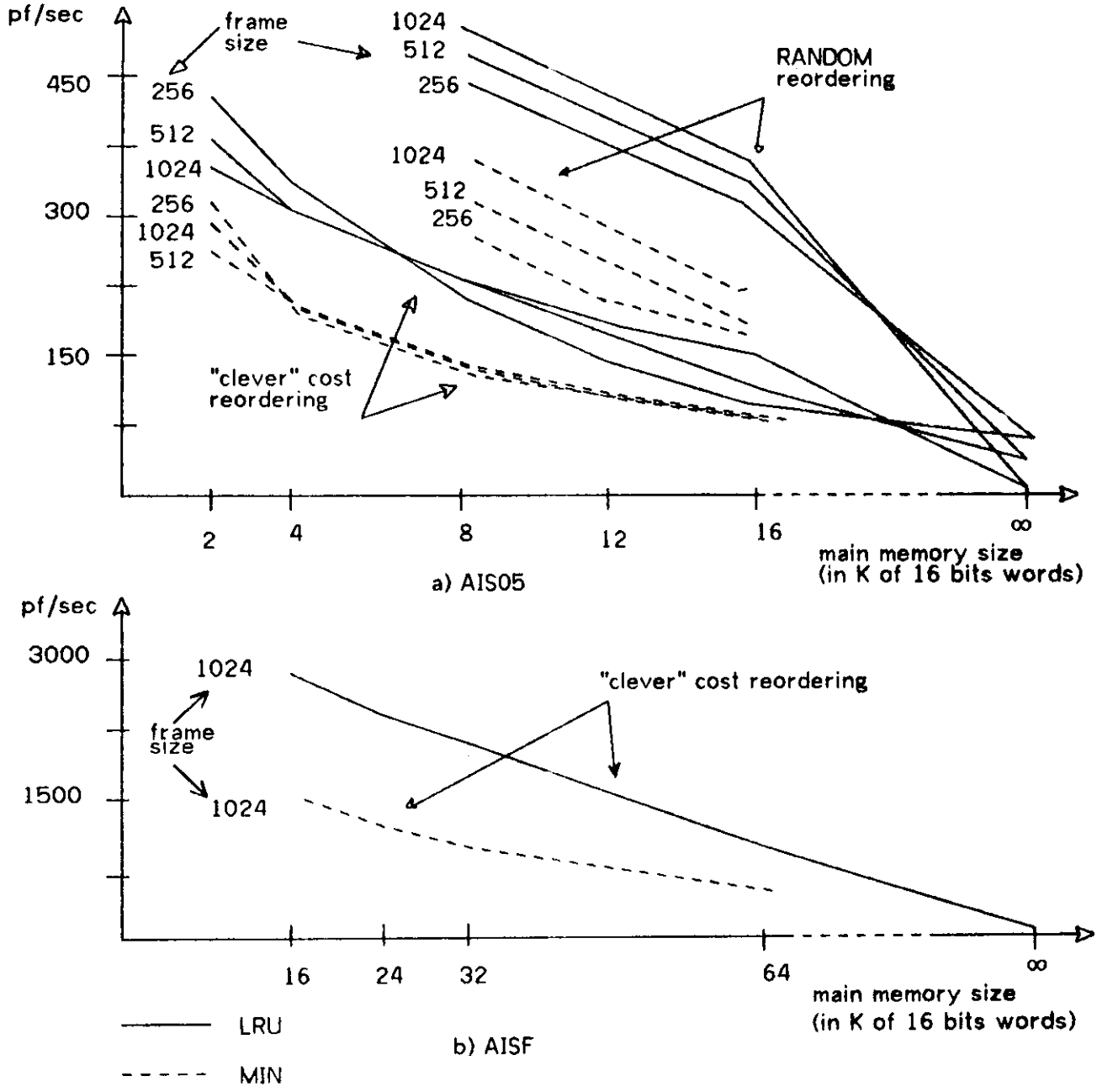


Fig.3 Page Faults/Second of Speech vs. Memory Size (Static Reordering)

policy uses a table that contains, for every page in the main memory, the time of the last access to the page and the time the page entered the memory. The search algorithm processes the states in bursts of time (segments) and all the states processed during a certain segment must be considered as accessed at the same time. This fact creates the problem of selecting the page to be replaced among all the least recently used ones. In this case our modified policy selects the page that least

recently entered the memory among the least recently used ones. Again, if more than one page is eligible for replacement, one among them is randomly selected.

Since we assume to be using a fixed size main memory we can use Belady's MIN algorithm [Belady, 66] to find the minimum number of faults with the given reordering. The MIN algorithm is also a good tool for measuring the effectiveness of a certain replacement policy. The replacement policy efficiency, defined as the ratio between the number of faults obtained with MIN and the number of faults obtained with a given policy, is a frequently used number. In our case the efficiency of LRU is average .6, this value is similar to the value of LRU efficiency in managing program references [Belady, 66].

Surprisingly enough the curves obtained do not present the characteristic knee generally observed in program behavior plots. This seems to imply that the search process is issuing references in a random way. This is not true at the state level since, given a set of states in the active list, only the successors of these states will be accessed in the next segment. Therefore we must conclude that the attempted reordering, though considerably better than a random reordering (see Fig.3), is not able to reproduce, at the page level, the locality of references that the search presents at the state level. As outlined in 2.1 this problem arises from the unpredictable behavior of the many different paths searched at the same time. We believe that this characteristic makes any static reordering unable to give a very good performance.

5 Proposed Techniques

The experimental results, obtained with the simulation of the secondary memory management policies previously described, show poor performance both in terms of number of page faults and in amount of main memory needed to obtain a given performance.

The main problem seems to be the impossibility of clustering the states in pages in order to take advantage of the block oriented structure of data on a secondary storage media. A lot of unuseful states are fetched along with the needed states not only increasing the secondary memory activity but, above all, wasting a lot of main memory space.

We tried to overcome this problem storing and retrieving in main memory variable length records containing all the information needed to process a state instead of fixed size blocks containing many different states. As outlined before (2.1), for the kind of networks we are considering, "to process a state" means to examine all the possible transitions from that state to its successors. Since each state has a variable number of successors, the record containing all the information needed to process a state is of variable length.

Managing variable length records instead of fixed length blocks requires a

significant processing overhead and this approach can be useful only if there is a substantial saving in terms of secondary memory accesses, in the case of a real time search, or in terms of main memory size, in the case of very large networks.

The secondary storage media currently available are block oriented. This implies that we will still have to cluster states in blocks in order to store and retrieve them from secondary memory. Moreover every time we need to retrieve a given state from secondary memory we are forced to read at least one full block that usually contains other states besides the state we are interested in; therefore we can choose to store in main memory all the states that are contained in the fetched block or just the currently needed state. As a matter of fact many different approaches are possible.

With regard to the problem of clustering states into blocks we have the following possible reordering policy:

- one block contains just one state;
- one block contains more states (as in the usual clustering policies);
- one block contains more than one state and a given state can be contained in more than one block (redundant reordering).

With regard to the problem of storing in main memory the states contained in the retrieved block we have the following fetching policies:

- the needed state is stored in main memory, the additional states retrieved are discarded;
- all the states in the retrieved block are stored in main memory;
- only the needed state is stored in main memory but the retrieved block is checked and any possible other useful state contained in that block is fetched.

The reordering policies and the fetching policies presented above can be combined to obtain many different strategies each characterized by a different number of secondary memory accesses and a different computational overhead. The following sections deal with a few simulation and implementation experiments that have been performed in order to analyze the behavior of the Harpy search under different secondary memory management strategies.

5.1 Single state fetch

The simplest fetching policy is to retrieve one state per block. The states can be stored one per block and the state names made to correspond with the block numbers, or they can be contiguously stored in blocks and their names made to correspond to the block number and the offset in the block. The simulation results for this approach are presented in Fig.4.

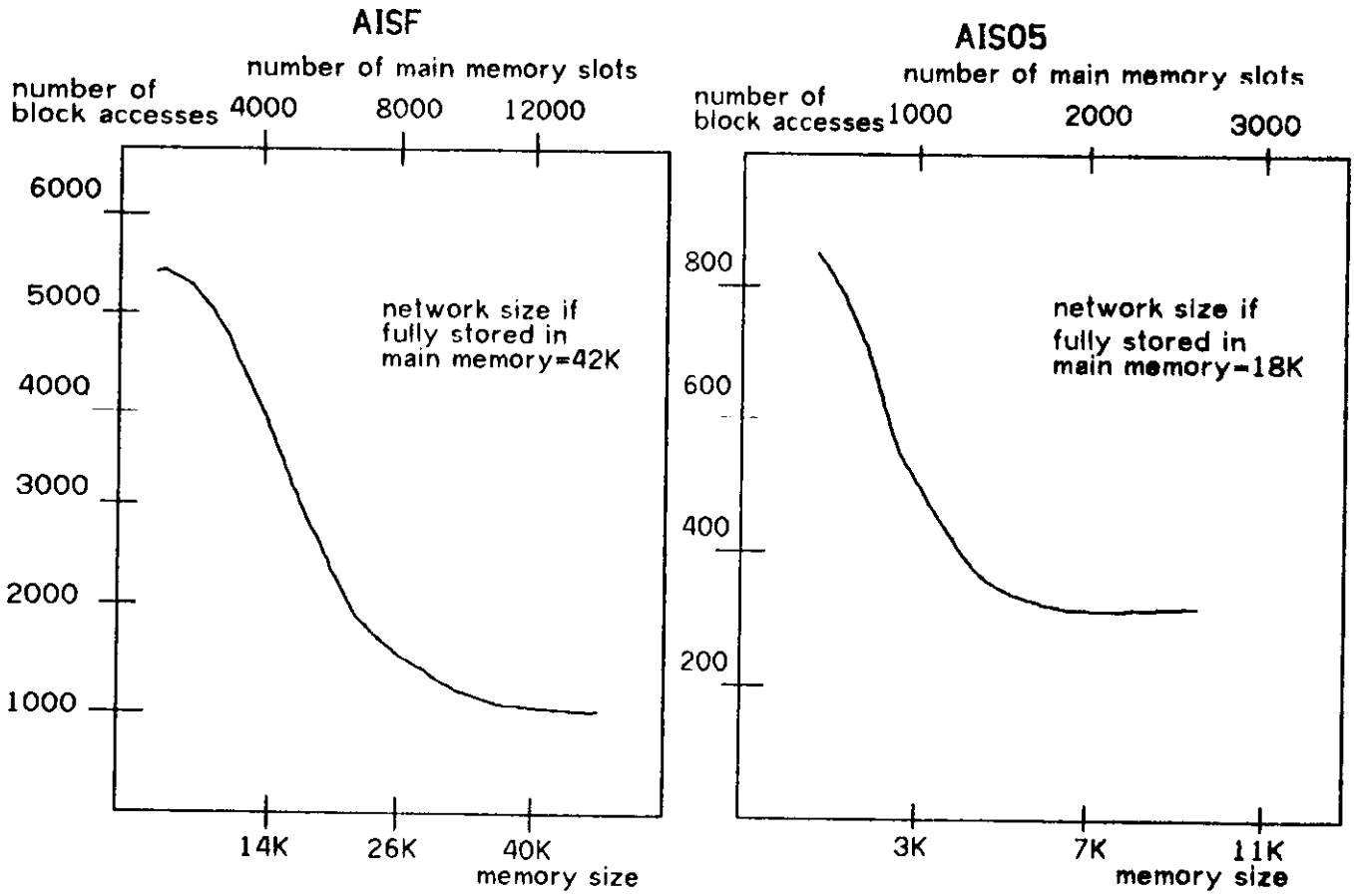


Fig.4 Performance of Harpy with the Single State Management Strategy

The number of page faults per second of speech is plotted against the number of "state slots" available in main memory. Each "state record" needs one slot for every successor of the state plus one slot for the state itself. Every time a state is needed and there are not enough free slots in main memory, sufficient least recently used states are replaced until the state record can be stored in main memory.

As explained in 3.1 the number of bits required for each state is 45. The effective number of bits required in a real implementation is actually more than 45 since, for the sake of efficiency, each slot must occupy an integer number of memory words. Moreover, supposing to store the records in contiguous locations, the "free slots" would be quickly scattered all over the memory, making it impossible to store a big state (fragmentation). To avoid fragmentation, slots can be linked together in a list, this list can also be useful to keep track of the recently used states. Of course we can trade space for computation time, i.e. fragmentation can be avoided with a periodic garbage collection. In a conventional single processor computer architecture this overhead must be avoided because the computational needs of the search algorithm are already high.

To ease the comparison between the "single state management policy" and the "static reordering policy" a possible memory size is indicated along with the number of slots. The memory size has been computed assuming to store each slot in two 16 bits words and to use one word for each slot to link it in a list; moreover we assumed to use a hash table containing one entry for each three slots (i.e. the state record is considered to need an average of three slots). This table is used to find the location of a state in main memory.

The curves present the "knee" that is typical of program behavior. The minimum number of page faults is reached for a memory size that is 80% of the network size (if fully stored in main memory) in the case of AISF, and 30% in the case of AIS05. Remember that, when the network is fully stored in main memory, the space it needs can be minimized.

The minimum number of block accesses is very high and does not allow a real time search. None of our goals is satisfied by this approach, but the overall behavior shows a better sensitivity to an increase in the size of main memory. A comparison between Fig.3 and Fig.4 shows that both AISF and AIS05 behave better with a static reordering.

The main drawback of the single state policy is the high number of block accesses. The advantage of this policy is its good sensitivity to the locality of search behavior.

5.2 A PDP-10 implementation

This experiment involved the implementation of a secondary memory retrieval mechanism for the Harpy speech recognition system, currently working on a Decsystem 10 (KA-10 and KL-10). The experiment consisted in an extensive modification of the original system that uses a main memory resident network. Reading a block of secondary memory in a time sharing system requires a long time because of the high system overhead. Therefore the number of block acceses must be minimized at all costs, even if more main memory and more processing time are needed.

The network was clustred in a redundant way: each state was stored at the beginning of a different block and the block was filled with all the successors of the first state. Therefore each block contained a subtree of the network and each state could appear in many different blocks as successor of different states.

The fetching policy that was used in this experiment stores not only the requested state in main memory, but also all the additional states contained in the retrieved block.

These reordering and fetching policies try to take advantage of the following characteristics of the search:

- due to the rich interconnectivity of the network some of the successors of the state just fetched can be needed in the same segment as parts of a different path;

- the successors of a state in the active list, if the state is not discarded, will be used in the next segment.

The performance of the system has been evaluated with the same data used for the simulations; the performance data is presented in Table I, it is not possible to make a direct comparison between Fig.3 and Table I because of the different size of memory words.

The number of block accesses is reasonably low, but the cpu overhead is high: the system requires about three times more cpu than the parent system that uses a main memory resident network. Moreover the amount of main memory used for each slot is very high, this is not a problem with the currently available networks, but could impare the use of the system in the case of extremely big networks.

5.3 Small locality strategy: A PDP-11 implementation

All the previous experiments show how difficult it is to exploit the locality of the search process. Both the static reordering and the single state management strategy need a high amount of memory and cpu time to give reasonable results. These strategies behave well enough for the current networks (although they are far from real time), but would probably be inefficient if applied to bigger networks.

The "small locality strategy", that we will describe in this section, was devised to reduce both the main memory and the cpu requirements. Actually the performance obtained with this strategy, measured on a real system, turned out to be superior to the more sophisticated strategies previously described.

The underlying idea is very simple: since the states in the current active list are all successors of the states in the previous active list, grouping together the successors of the same state creates small "clusters" that contain states having a high probability to be processed during the same segment. No assumption is made on the relationship between different clusters. Therefore this strategy only partially exploits the locality of the search process.

The "small locality reordering" can be obtained with a very simple algorithm:

a) the initial state is stored in a list called "next states list";

b) all the states in the next state list are stored in the secondary memory;

c) the successors of the states being stored on the secondary memory are saved, if they have not been clustered already, in the next states list;

d) steps b and c are repeated until the next states list is empty.

For example, the reordering generated for the partial network of Fig.1, assuming that the next states list contains ABCD when this part of the network is examined, would be ABCDOEFHLMPNR.

During the reordering the states are renamed with the block number and offset in the block of their physical location, this allows a quick and efficient retrieval since no mapping table is needed.

During the search the states in the active list are sorted at the beginning of each segment. States are not kept in main memory but are retrieved each time they are needed.

Every time the disk is accessed, more than one block is retrieved (the retrieval of a limited number of consecutive blocks from a disk does not require much more time than the retrieval of a single block).

The reordering and retrieval strategies described limit the number of block accesses required because, for each disk access, more than one state is processed.

This strategy has been implemented on a PDP-11/40 running under the UNIX time sharing system, the disk was an RP06. The use of a minicomputer gives the possibility of easily and efficiently implementing the required access of multiple sequential blocks with only one disk seek. The cpu time required for the secondary memory access management is about 10% of the time spent in executing the search algorithm (as opposed to 200% in the PDP-10 experiment). The number of block accesses is highly reduced by this strategy.

Fig. 5 shows the number of block accesses per second of speech vs. the number of contiguous 256 words blocks retrieved with each disk access (or the number of memory buffers required). Using only one buffer gives already an improvement of about 50% over the single state fetching policy (Fig.4). Every time the number of buffers is doubled the number of disk accesses decreases of about 30%. The other curves in Fig. 5 represent the time spent by the system in reading the disk, this time also decreases as the number of buffers increases. The percentage of improvement of the disk access time decreases as the number of buffers increases, because the time required to read the blocks partially balances the improvement obtained reducing the number of disk accesses. It was not possible to examine the behavior of the system with more than 16 buffers because the memory required was not available.

5.4 A solution using bubble memories

All the previously presented results fail to attain a real time retrieval of the network information. This is partially due to the limited speed of the available rotating media and partially to the difficulty in efficiently managing the states in fixed size blocks (as required by the currently available secondary memory systems).

A few technological alternatives are possible or will be possible in the next few years, namely: charge coupled devices (CCD), electron beam memories (EBAM) and bubble memories (BM). CCDs are fast enough but their price at present is not too far from the price of random access memories, this makes them too expensive for big

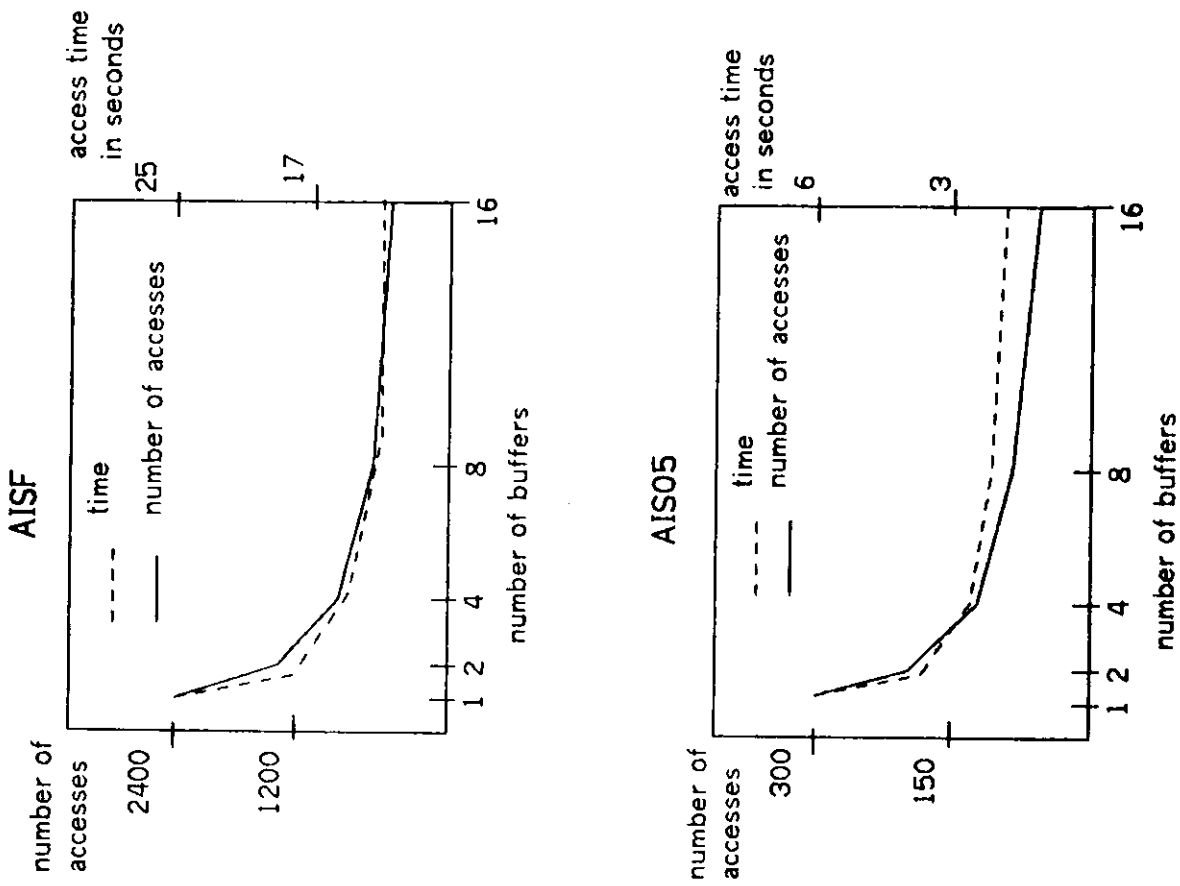


Fig.5 Performance of Harpy with the Small Locality Strategy on a PDP-11/40

networks. EBAMs probably represent the right solution but must be very big (more than 10^8 bits) to have a favourable price, they are therefore useful only for extremely large networks. The price performance ratio of BMs meets very well the needs of today's speech recognition systems. Moreover BMs present a few unique characteristics that enhance their performance in the particular case we are considering: they are cyclic memories that can be stopped and (not in all implementations) made to rotate bidirectionally. At some future date, CCDs may be equally as attractive as BMs.

Moreover they are usually produced in chips that can be conveniently paralleled to increase the bandwidth.

We simulated the behavior of one of the few currently available bubble memory chips, the Texas Instruments TBM0101 [Texas, 77]. This chip presents a few drawbacks for our application:

- its architecture makes it impossible to shift the bubbles backwards;

- the speed at which the bubbles circulate is not very high.

		HARPY with network in core	HARPY with network in sec. memory	HARPY with network in sec. memory	Bubble memory system
AIS05 for every second of speech	number of secondary memory accesses	—	101	69	—
	secondary memory access time	—	—	3	.79
	cpu time	—	—	7.8	—
	elapsed time	10.1	14.6	10.6	—
AISF for every second of speech	number of secondary memory accesses	—	850	339	—
	secondary memory access time	—	—	7	5.1
	cpu time	—	—	43.2	—
	elapsed time	40.4	196	50.2	—
		PDP-10 (KA-10)		PDP-11/40	

Table I Performance Comparison of the Different Harpy Systems Described

The architecture that has been simulated uses 45 bubble chips (one for every bit of the state information) circulating in parallel. This organization allows a very high transfer rate (2.25 Mbits/sec) so that a state with three successors can be transferred in 80 microseconds. The theoretical average and maximum access times are, respectively, 4 ms and 7.2 ms. Therefore, supposing to read 5500 state records for every second of speech (Fig.4), the retrieval time would be 22 seconds per second of speech. If we take into account the possibility to stop the bubbles after each access the retrieval time drops to 5.1 seconds per second of speech (Table I). Although this is not real time yet, it is reasonable to assume that, with the technological improvement that is foreseen in the field of bubble memories, it will soon be possible to implement a secondary memory architecture that allows a real time retrieval of an AI network.

6 Conclusions

The problem of efficiently storing and retrieving a special class of knowledge data bases, namely knowledge networks, has been tackled. The analysis of the search process and a few experiments show that the reference patterns generated are different both from usual data bases references and from program references.

Networks references show a good locality and therefore the main memory can be efficiently managed with the usual replacement policies. On the other hand knowledge networks are not suitable for dynamic restructuring techniques because of the data dependence of their reference patterns. A static reordering has been simulated, the performance data obtained show that it is very difficult to cluster the states in blocks in a way that exploits the locality of the search process references. This problem can be partially overcome by retrieving from secondary memory single states instead of fixed size blocks. This strategy, called "state level memory management", lowers the number of secondary memory accesses but also requires a high amount of processing time. This is shown by the measurement of the performance of the Harpy speech recognition system implemented on a Decsystem 10.

To overcome these problems a different strategy, called "small locality strategy", was implemented on a PDP-11/40 running under the UNIX time sharing system. This strategy makes use of a very simple network reordering algorithm that tries to cluster the successors of the same state together. The states are not stored in main memory but are processed as they are retrieved. Although the required amount of main memory and cpu time are very low this strategy outperforms the previously described strategies.

All the described experiments fail to attain a real time retrieval of the network information, this is partially due to the hardware limitations of the currently available secondary memory systems. Thus the technological alternative of bubble memories has been investigated. The simulation of a secondary memory system using one of the few currently available bubble memory chips gave good results. It is believed that, with the technological improvement that is foreseen in the field of bubble memories, it will soon be possible a real time retrieval of an AI network.

Acknowledgements

Many of the ideas presented in this paper have been influenced by discussions with Raj Reddy, Bruce Lowerre and Ken Greer. Bruce Lowerre implemented the modified version of the Harpy system on the Decsystem 10. Ken Greer implemented the PDP-11/40 version of the Harpy system.

7 References

[Baier and Sager, 76] J.L. Baier and G.R. Sager "Dynamic Improvement of Locality in Virtual Memory Systems" IEEE Trans. on Soft. Eng. Vol.SE-2, no. 1, March 1976.

[Belady, 66] L.A. Belady "A Study of Replacement Algorithms for a Virtual Storage Computer" IBM Systems Journal 5,2 1966.

[Comeau, 67] L.W. Comeau "A Study of the Effect of User Program Optimization in a Paging System" ACM Symp. on Operating System Principles 1967.

[Denning, 68] P.J. Denning "The Working Set Model of Program Behavior" CACM Vol.11 No.5 May 1968.

[Ferrari, 74] D. Ferrari "Improving Locality by Critical Working Sets" CACM Vol. 17 no. 11 November 1974.

[Ferrari, 76] D. Ferrari "The Improvement of Program Behavior" Computer November 1976.

[Hatfield and Gerald, 71] D.J. Hatfield and J. Gerald "Program Restructuring for Virtual Memory" IBM System Journal Vol 10 1971.

[Lowe, 70] T.C. Lowe "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing" CACM Vol. 13 No. 1 January 1970.

[Lowerre, 76] B. Lowerre "The HARPY Speech Recognition System" PhD Thesis Computer Science Department, Carnegie-Mellon University, 1976.

[Ramamoorthy, 66] C.V. Ramamoorthy "The Analytic Design of a Dynamic Look-Ahead and Program Segmenting System for Multiprogrammed Computers" Proc. ACM National Conference 1966.

[Texas, 77] Magnetic Bubble Memories and System Interface Circuits from Texas Instruments. February 1977.

[Ver Hoef, 71] E.W. Ver Hoef "Automatic Program Segmentation Based on Boolean Connectivity" Proc. AFIPS 1971 SJCC.