STATISTICAL MEMORY MANAGEMENT FOR DIGITAL SIGNAL PROCESSING

by

Farshid Moussavi

Memorandum No. UCB/ERL M91/68

23 July 1991

STATISTICAL MEMORY MANAGEMENT FOR DIGITAL SIGNAL PROCESSING

by

Farshid Moussavi

Memorandum No. UCB/ERL M91/68

23 July 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

STATISTICAL MEMORY MANAGEMENT FOR DIGITAL SIGNAL PROCESSING

by

Farshid Moussavi

Memorandum No. UCB/ERL M91/68

23 July 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

STATISTICAL MEMORY MANAGEMENT FOR DIGITAL SIGNAL PROCESSING

Farshid Moussavi

Professor David G. Messerchmitt

Department of Electrical Engineering and Computer Sciences

UC Berkeley

ABSTRACT

As processing throughput increases thanks to advancements in architecture and technology, memory bandwidth has increasingly emerged as a bottleneck in many digital signal processing tasks. In most cases so far, application specific solutions were found to improve memory bandwidth. Where locality existed, small amounts of cache would pay off significantly [24]. In other cases where high bandwidth was needed only for short periods of time and accessing was predetermined, lookahead techniques proved useful. Sometimes when neither of the above held, the entire main memory was implemented in fast(and expensive) technology. This study looks at memory interleaving as anarchitecture level solution to the main memory bandwidth bottleneck. By properly allocating data items in various banks, significant speedup can be achieved. Four algorithms for data allocation are presented and analyzed. The improvement achieved by these algorithms is shown to be a function of various system parameters, e.g. number of banks, bank size, and cache size; as well as the nature of the memory access. Of these four algorithms, two are shown to be particularly useful due to their superior performance, simplicity, and utility for real time applications.

Table of Contents

STATISTICAL MEMORY MANAGEMENT FOR DIGITAL SIGNAL PROCESSING

Farshid Moussavi
Professor David G. Messerchmitt
Department of Electrical Engineering and Computer Sciences
UC Berkeley

1. INTRODUCTION

Recent years have witnessed dramatic increases in processing speeds thanks to advancements in architectures, algorithms, and technology. With these improvements, newer and more complex digital signal processing tasks have become feasible [1-5]. In some of these tasks, reasonable amounts of data need to be handled, or there at least exists some locality in memory access, hence data can be accessed quickly and efficiently. However, in many applications such as speech and video where large amounts of data are handled, memory bandwidth quickly emerges as the bottleneck. Various solutions have been used, including the use of large amounts of fast and expensive technology to implement main memory (brute force solution). Where locality existed in the data access patterns, memory hierarchies and cache have been very helpful [6,7,8,24]. Unfortunately however, in some applications, e.g. speech, none of the above holds. In such cases, an efficient scheme is needed to access a large main memory quickly if the design is to remain economical.

This study looks at memory interleaving as a solution to this problem. By allocating data items to the right banks, and allowing for possible duplication, considerable speedup can be achieved without the use of faster, more expensive technology. This idea is particularly suitable for special purpose applications (such as speech recognition) where the data access pattern statistics are more predictable. Several algorithms are proposed for the proper allocation of data items, and the results of these algorithms are presented and analyzed.

The remainder of this report is organized as follows. Section 2 outlines and describes a problem in speech recognition which motivated this study. Section 3 presents the memory interleaving approach as a potential solution to this problem. In section 4, several algorithms for data allocation in an interleaved memory system are described in detail. The results of using these algorithms in different situations are presented in section 5. Finally, section 6 summarizes and concludes the report.

2. MOTIVATION AND BACKGROUND

2.1. Introduction

The original motivation for this study came from a problem in speech recognition. The particular system involved was a 20,000 word real time continuous speech recognizer developed by IBM. We wanted to find ways to speed up a certain task within this system. The bottleneck of this task was memory bandwidth. After some thought, we decided to look at the problem in a more general sense. In this section, we will present the original problem to illustrate a potential application of our techniques.

2.2. Speech Recognition System

The system, whose block diagram is shown in figure 1, is composed of an acoustic processor and a linguistic decoder [9,10,11]. The acoustic processor receives speech and encodes it into a sequence of acoustic labels, y, using LPC analysis and vector quantization [16,17]. The linguistic decoder conducts a maximum likelihood search to find the most probable word string w, given the acoustic label string y was received. This search is based on Hidden Markov Models (HMM's) of the words, and the apriori probabilities of the words having been uttered, known as the language model.

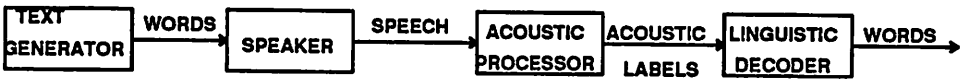

Figure 1 (prop31). Speech Recognition System Block Diagram

2.2.1. The Linguistic Decoder

The linguistic decoder uses a stack algorithm to recover the original word string. In this algorithm, the acoustic label string is segmented into substrings, and each substring is matched to a word based on a maximum likelihood search. Naturally, the segmenting of the acoustic labels may need to be changed and backtracking is required.

Even for a moderate vocabulary size, searching through the entire vocabulary for the most probable word would be an enormous task. To overcome this problem, there is a search hierarchy with three levels of pruning (See figure 2). First, the *fast match* preselects 500 words which have similar initial phones to those indicated by the given acoustic label substring. The second level of pruning comes from the *language model*, which selects words based on their probability of having been spoken in the English language. The language model narrows the search down to 50 words. Finally, the *detailed match* calculates $P(y|w)$, the probability that the acoustic label substring y was created given the word w was uttered, for each of the 50 words being searched. $P(y|w)$ is also called the *acoustic match*, and is needed in the main algorithm to conduct the maximum likelihood search for the most probable word string [9,10,11].

Therefore, the detailed match can be thought of as a subroutine called upon by the main algorithm to calculate acoustic matches for different acoustic label substrings. Meanwhile, the candidate words to be searched for each computation are fed in from the fast match and the language model, which prune the search to reduce the computational load.

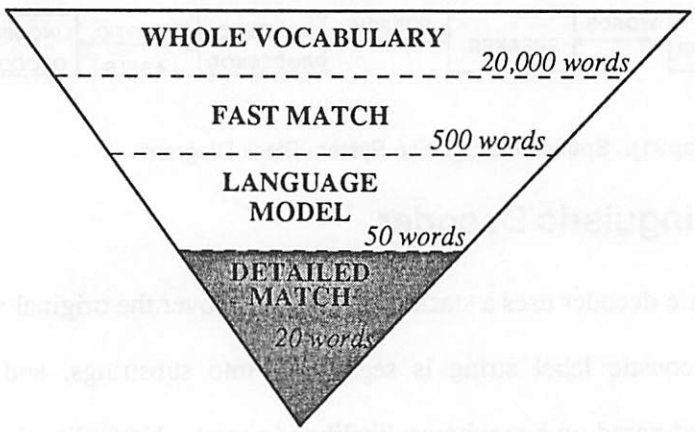

Figure 2 (hierarchy). Hierarchy for Pruning the Search

Our original concern was to find ways to speed up the operation of the detailed match, which happens to be bounded in speed by memory bandwidth, as we will see shortly.

2.2.2. Operation of the Detailed Match

As mentioned before, the detailed match takes an acoustic label substring from the main algorithm, and 50 words proposed by the language model, and calculates $P(y|w)$, the acoustic match for each of those 50 words. It returns this information for 20 of the most likely words to the main algorithm, which continues its maximum likelihood search.

The calculation of the acoustic match for each word is based on the word's Hidden Markov Model (HMM). The Hidden Markov Model is shown in figure 3. It is very similar to the regular discrete parameter Markov process. The difference is that outputs, not states, are observed (hence the word "hidden"). There is a probability mass function associated with each state that determines the outputs which occur in that state. In HMM speech recognition, the outputs are acoustic labels [10-12].

Each word's model has roughly 100 states, which may vary for different words. These states may be organized into a column, shown in figure 4. The word models are left to right,

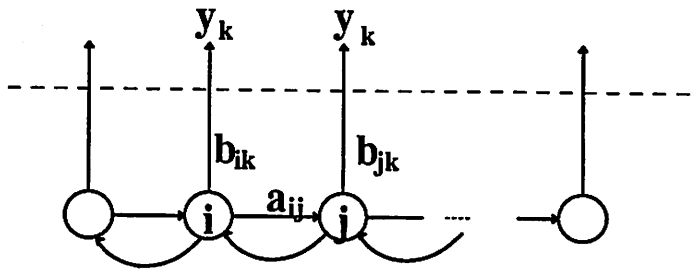

Figure 3 (HMM1). Hidden Markov Model

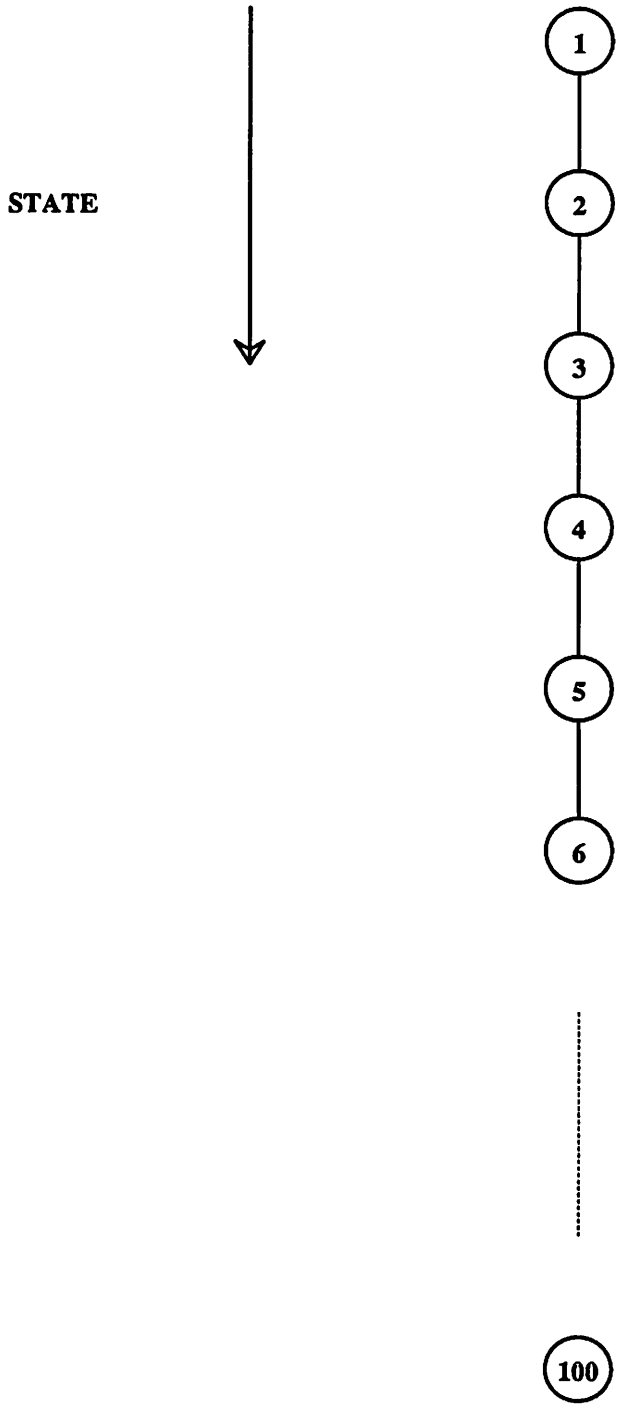

Figure 4 (dmatch1-3). One Column of States in the Trellis.

i.e. transitions are allowed only from higher states to states of the same or lower level. For each transition, there is a transition probability, and an acoustic label output. This output is

specified by a probability mass function. For each state, there are as many output probabilities as acoustic labels. In this particular system, there are roughly 200 acoustic labels.) [10,11] At each time increment, the states are updated as shown in figure 5. To simplify the transition structure, state transitions from higher to lower states are allowed at a given time. However, such a transition creates no output. These transitions are called *null transitions* [10], and are depicted by dotted lines. The resulting transition structure is shown for one time increment in figure 6.

Based on these concepts, a trellis structure can be constructed for each word with time as the horizontal dimension and state as the vertical dimension, as shown in figure 7. In this structure, each state is the target and the origin of 3 transitions. It is also possible for any state to reach a state of equal or lower level in the next time increment. The trellis is completely defined if the transition probabilities and the output probabilities are defined for all states.

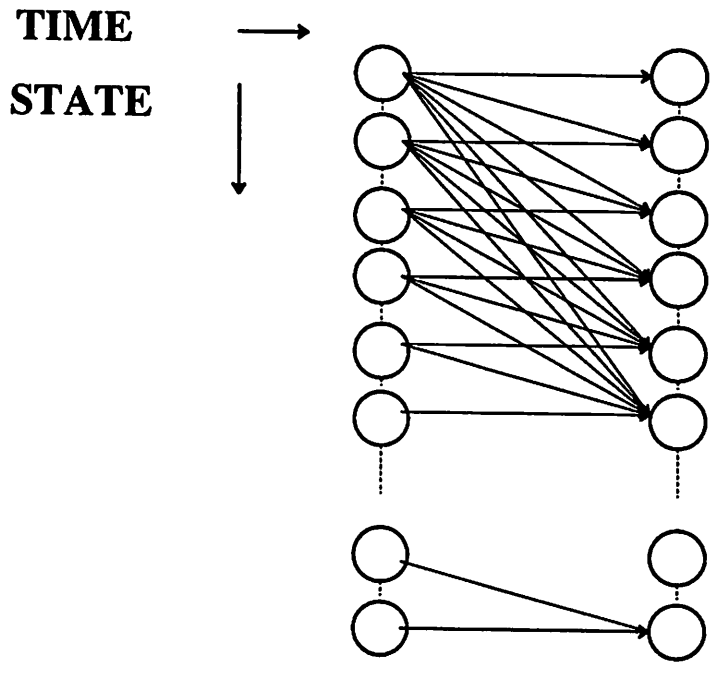

Figure 5 (dmatch1-4a). Updating the Column of States for One Time Increment.

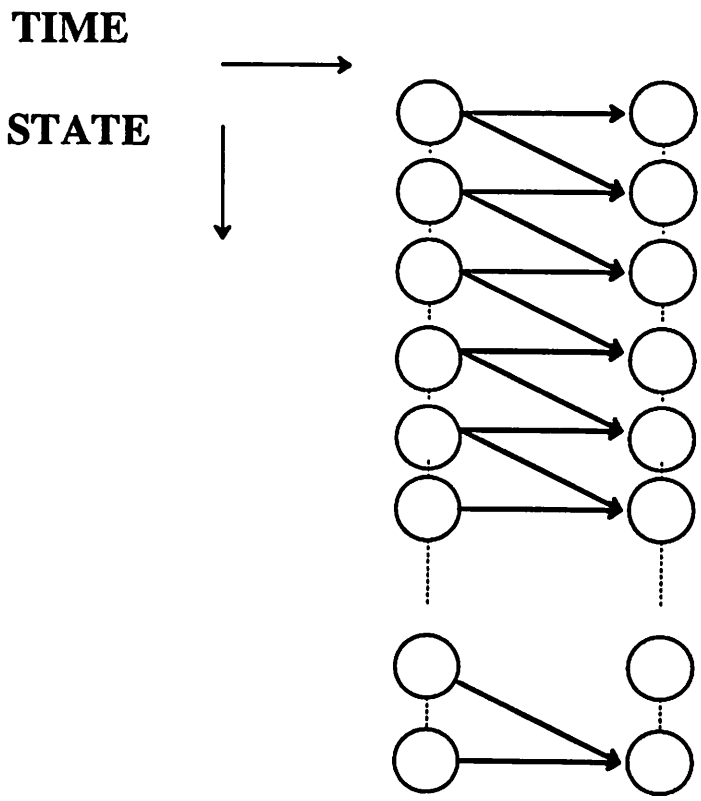

Figure 6 (dmatch1-4b). Modified Structure for Updating the Column of States.

The detailed match calculates the acoustic match as follows. For each word proposed to the detailed match, it looks up the word's HMM states and model data. Therefore, it constructs one trellis for each candidate word. At the left side of the trellis there is a starting distribution obtained from the previous trellis calculation. The detailed match then uses the transition probabilities and the output probabilities to calculate the probabilities of each state recursively from those of its surrounding states. The necessary transition probabilities depend only on the word, whereas the necessary output probabilities depend on the word and the current acoustic label. The required data is stored in lookup tables. The calculation is terminated once the end of the acoustic label substring is reached.

At this point, the detailed match calculates the probability that that the acoustic label substring y was observed. This quantity is $P(y \mid w)$, the acoustic match. Looking at the trellis

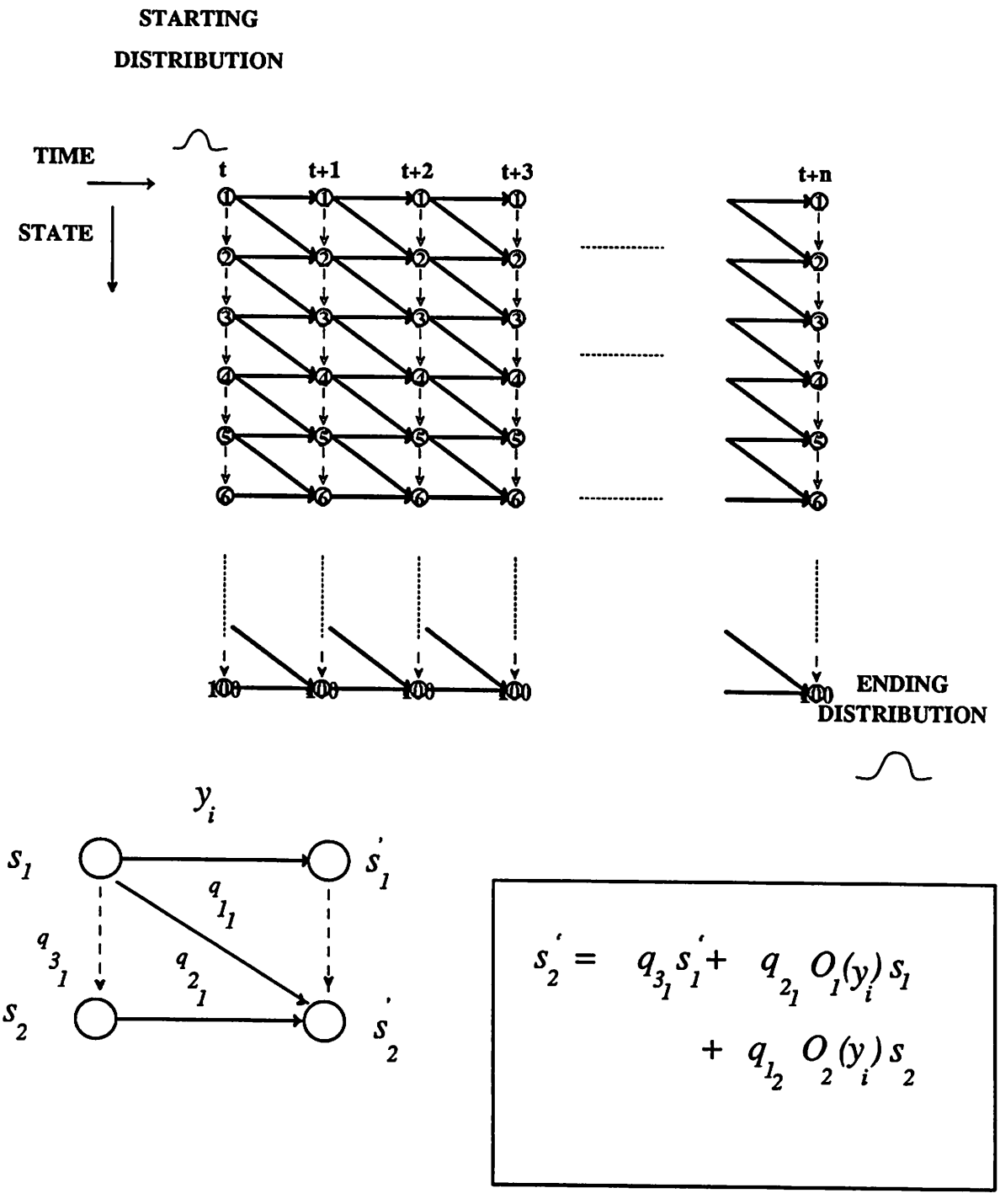

Figure 7 (trellis). The Trellis Used by the Detailed Match for Computation of the Acoustic Match.

structure, and neglecting some paths which were discarded because of their low probability, it

can be seen that for each path from the initial state to the final state, there exists a possibility that the acoustic label sequence y resulted from the word w. The probability for this event is the product of the transition probabilities and the appropriate output probabilities along the path from the initial state to the final state of the trellis. Summing this probability over all possible paths from the initial to the final state yields $P(y \mid w)$, the acoustic match.

For this calculation, 300 transition probabilities are required, which can be initially loaded and reused throughout the trellis computation. In addition, for each time increment, 200 output probabilities are required and must be accessed from main memory. The memory location of these probabilities depend on the candidate word (the choice of states), and the acoustic label at that given time. The structure of the memory containing output probabilities is shown in figure 8. Therefore, for each word whose acoustic match is to be calculated, the appropriate states are found. Then, during each time increment in the trellis computation, the output probabilities for these states and the acoustic label in the given time increment are accessed. This continues until the end of the trellis is reached, at which time a new word is put forward, and different states are chosen. This is illustrated in figure 9, which shows

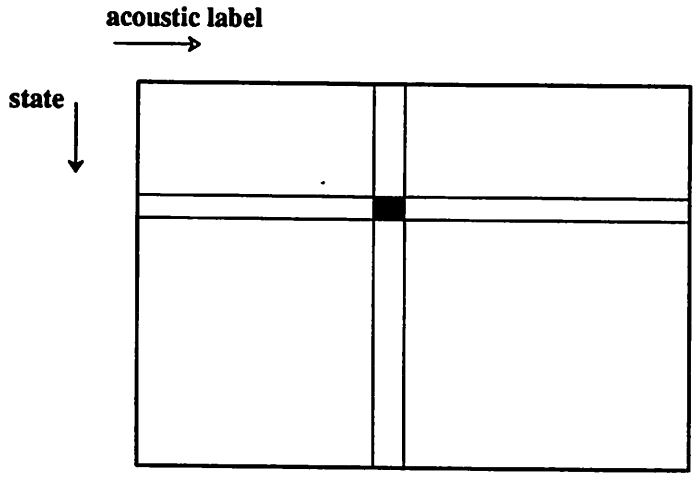

Figure 8 (dmatch_org). Output Probability Memory Organization

"snapshots" of the output probability memory access. The access of output probabilities is the memory bottleneck of the detailed match. Since there is not necessarily any temporal locality in the arrival of the acoustic labels, a cache would be of little help here. Also, since there are a large number of states and each one has 200 output probabilities, the memory required for the output probabilities is huge. To implement this in a fast technology would not be economical. Therefore, an architecture level solution must be found to improve the speed of this procedure.

In this study, we propose memory interleaving as a solution to this and similar types of problems. By dividing main memory into R banks and interleaving the access, a worst case speedup of unity and a best case speedup of R are achieved. If duplication is introduced, overall speedup can be improved. For a desired speedup S, duplication need be at most S. It might be possible to achieve close to such a speedup with considerably less duplication, if allocation is done in a clever way. This problem is introduced more formally in the next section.

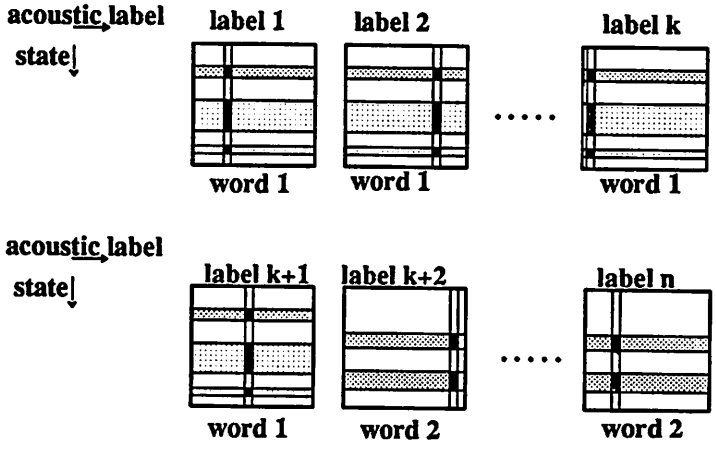

Figure 9 (dmatch_access). Snapshots of the Output Probability Memory Access

3. THE MEMORY INTERLEAVING APPROACH

3.1. Introduction

This section introduces and formulates a memory interleaving problem for high speed digital signal processing systems which are bounded by the system's memory input/output bandwidth. Specifically, the problem of data allocation to different memory banks based on data access statistics is addressed. The solution to this problem could be useful for a variety of applications, such as speech recognition, video communications and signal processing, vector quantization, and any other memory i/o bound DSP problem.

3.2. Problem Formulation

The problem to be solved is shown graphically in figure 10. There is some processing element which performs operations at a high speed. The operations require the accessing of reference data from main memory, i.e. there needs to be a steady flow of data from main memory to the processing element. There is also a fast cache memory capable of operating at the processor speed. It is assumed that the processing speed is several times higher than the main memory speed. Accessing only one memory bank by the processor for each required data item would create a bottleneck. One way of increasing the rate of data flow is to organize main memory into several banks, each containing some of the data items of main memory, and to interleave the accessing of data items from these banks.

Assume there are R banks of memory. The goal is to speed up the data flow rate as much as possible by properly distributing data items among these banks. The speedup achieved cannot be larger than R unless cache is used. For the case where the data item sequence to be accessed is predetermined for all time, the placement of data items into the R banks is trivial. However, for random data sequences, a scheme is needed to allocate data to the various banks based on the data sequence's statistics.

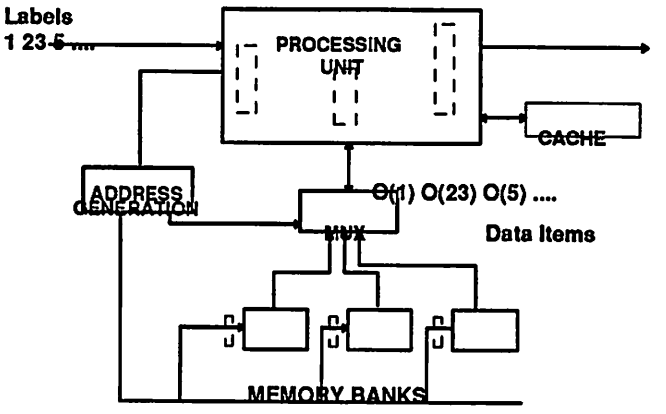

Figure 10 (memint). A System Using Memory Interleaving.

The data in the speech recognition problem was organized into columns according to a natural structure of the computation (the structure of the trellis). In general, this may not be true, and data might not be previously organized into columns. However, for this study, we assume that data has previously been organized into columns by some scheme, and it is the *columns* of data, not the individual data, that we seek to allocate. The reason for this is that in any practical implementation of memory interleaving, there must be some mechanism of keeping track of the locations of the data to be accessed. The locations of a relatively small number of columns can be kept track of much more easily than the locations of all the individual data. In fact, keeping track of the locations of individual data would require a separate memory at least as large as the main memory itself, which is unacceptable.

Following is a more detailed formulation of the problem:

Given:

A computational system which uses memory interleaving to access data in a random manner,

A set of R memory banks B = $\{b_1, b_2, \ldots, b_R\}$;

An alphabet A representing *l* columns from which data items are accessed randomly A = {1, 2, ...*l*} (The members of A are called *labels*, and the columns referred to by the labels are called *data columns*, or simply *columns*. The assumption is that all the data has already been pregrouped into these columns, and the label determines the column access.) ;

A "training sequence" of data accesses which are from A and which are statistically representative of real data accesses by the system;

A ratio of processing speed/memory speed = P;

A desired memory access speedup S, which is upper bounded by Min(R,P) (except when cache is used);

A maximum allowable cache size c; and

A maximum allowable amount of overall duplication, i.e. actual overall memory size/minimum memory size, d.

NOT Given:

Any probabilistic information about the process which generates the sequence for data to be accessed, except that it is possibly stationary in the short term (as in speech).

Wanted:

A mapping, or assignment M: A ---> B (allowing for duplication) such that the desired speedup S is achieved. In other words, we would like to divide the set A into (possibly overlapping) subsets b_1 through b_R such that the probability of any b_i occurring less than P increments of time after its last occurrence is minimized (See figure 11.)

To illustrate the problem further, assume the main memory organization shown in figure 12. Both row and column accesses are random. Now, divide the memory into columns, as shown in figure 13. Every time a label arrives, a subset of the corresponding column must be accessed. This subset changes with time. A good example of this is the access example in

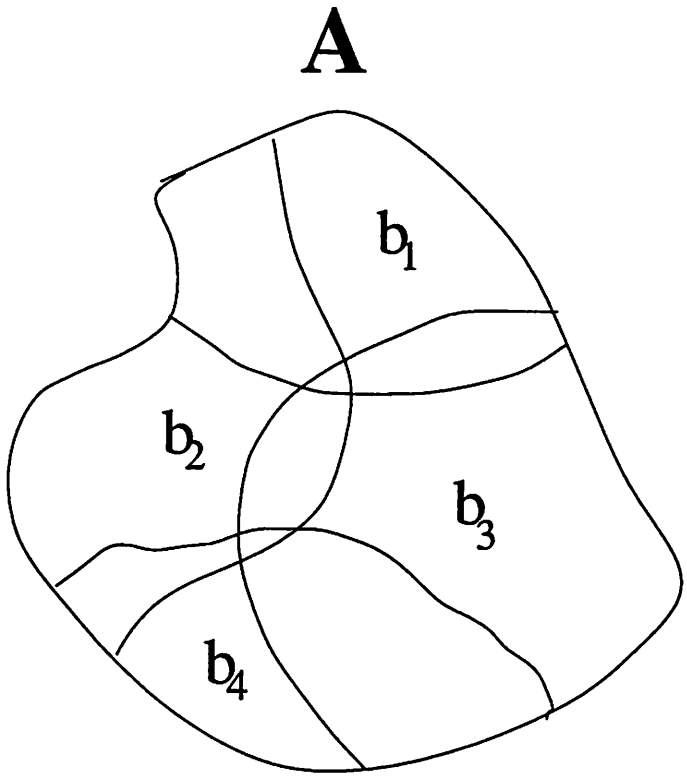

Figure 11 (partition). Partitioning of Main Memory into Possibly Overlapping Subsets

figure 9. Now, by splitting main memory into smaller banks, and interleaving the accessing of these banks, we hope to increase memory bandwidth. The division is made only along the column boundaries, and columns can be used more than once, as long as the overall memory size limitation is not exceeded.

$$M_{i,j} = \begin{cases} 1 & \text{if } \textit{data item } j \textit{ is in } b_i \\ 0 & \textit{otherwise} \end{cases} \qquad (2)0$$

For example, if bank 1 were to contain columns 1 and 3, the first row of M would be [1 0 1 .....]. It can be seen that the total number of columns stored in main memory would be $\sum_i \sum_j M_{i,j}$ and the total duplication d would be $\sum_i \sum_j M_{i,j}/l$. In this context, the problem can be viewed as designing the matrix M to minimize contention between banks when banks are accessed at the processor rate.

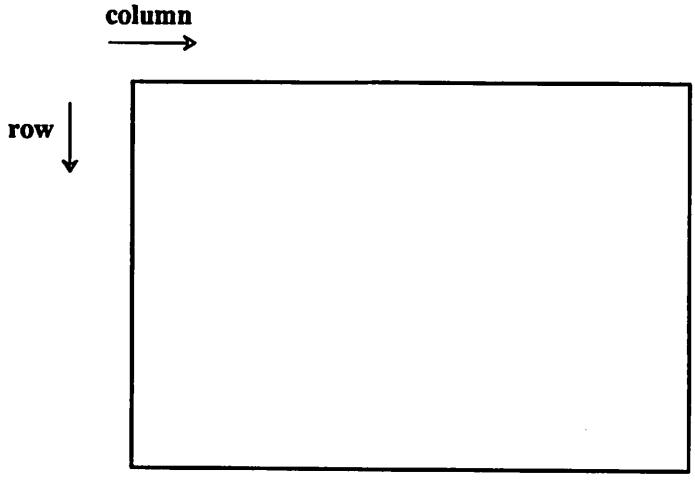

Figure 12 (mem_org1). Typical Organization of a Main Memory

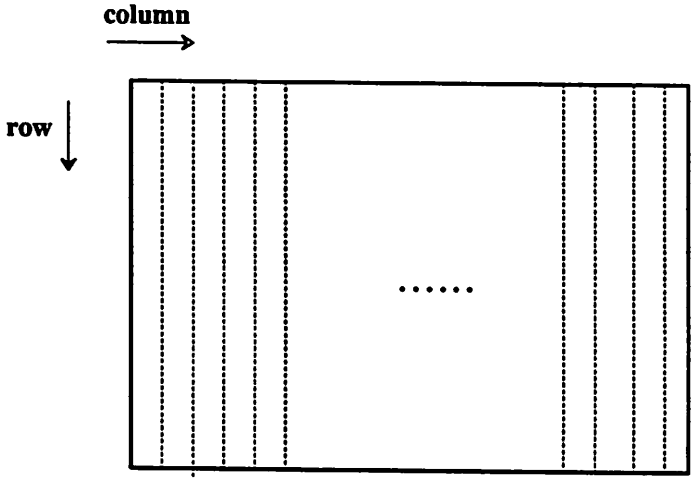

Figure 13 (mem_org2). Division of Main Memory into Columns

In this study, for the purpose of address space efficiency and simplicity, we shall limit ourselves to cases where the number of banks is a power of 2.

Before attempting to solve this problem, several facts must be noted:

1. In general, for an analytical solution, the S^{th} order statistics of the process are needed. Since we are interested in cases where S is larger, duplication is allowed, and no probability information about the process is assumed, the problem is extremely difficult to solve

analytically.

2. Different solutions may pay attention to the banks' average access behavior through all time, their instantaneous access behavior, or both.

3. $P(b_i)$ is the probability that b_i is accessed. The sum over all b_i's of this quantity must naturally be unity. A similar condition exists for $P(j)$, the probability that an item from column j is requested. Therefore, it can be written that

$$\sum_{i=1}^{R} P(b_i) = \sum_{j=1}^{l} P(j) = 1 \qquad (3)(I)$$

It should also be noted that the probability of any given bank being accessed is less than or equal to the sum of the probabilities of the member items of the bank being accessed. This can be represented by:

$$P(b_i) \leq \sum_{j=1}^{l} M_{i,j} P(j) \qquad (4)(I)$$

The equality holds if there is no duplication, i.e. if each column is allocated only to no more than one bank.

4. If each memory bank contains m data columns, the banks can be viewed as a matrix of Rm columns, which are rearranged, or skewed from their original pattern. If R columns are to be accessed without waiting, no more than R processor cycles are required. These columns constitute a vector. There are $\binom{Rm}{R}$ possible R vectors, whereas any skewing scheme will allow us to access at most m^R distinct vectors. It can be shown using Stirling's formula that the ratio of the total possible number of vectors to the number of vectors we can access without waiting is approximately [13]:

$$\left[\frac{1}{2\pi R}\right]^{\frac{1}{2}} e^R \qquad (5)()$$

which is greater than unity. Therefore, it is impossible to avoid waiting cycles altogether, since arbitrary R vectors cannot be accessed without wait states.

5. It can be shown that if $R \leq P$ and if periodic access is possible, periodic bank access facilitates maximum speedup, that is, minimizes the number of wait states. Of course, since any arbitrary R vector cannot be acquired in R cycles, wait states in general are unavoidable. Also, memory is most efficiently utilized when the probability of access of all memory banks are equal.

The approaches that we considered generally fall into two categories. The first looks at overall average access behavior, e.g. P(3), and ignores instantaneous and temporal behavior, e.g. P(3,9). The second category pays attention to instantaneous activity, and tries to minimize contention. The first class is simpler, and gives a better idea about how much duplication is required. The second class is more complicated (higher order statistics required), but gives a better idea of how data should be partitioned.

In the next section, we introduce four algorithms we investigated. These algorithms fall into either or both of the above categories. We then will present results of the simulation of these algorithms and their significance.

4. ALLOCATION ALGORITHMS

4.1. Introduction

In this section, we present four algorithms that we investigated. We also show a "generic" allocation algorithm for comparison purposes. The first algorithm only looks at first order statistics. The next three all pay attention to instantaneous access behavior, and therefore fall into the second category. We start the descriptions with the generic algorithm, which is the simplest of all.

4.2. Generic Algorithm

The generic algorithm is presented merely for comparison purposes. In this algorithm, the banks are filled in with columns corresponding to successive labels in the alphabet. When the end of the alphabet is reached, the allocation starts over from the beginning of the alphabet. Also, any time a bank is filled, allocation resumes at the beginning of the next bank. This process repeats until all banks are full. Here is an example for l = 16, d=2, and R=4:

b_1 = {1,2,3,4,5,6,7,8}

b_2 = {9,10,11,12,13,14,15,16}

b_3 = {1,2,3,4,5,6,7,8}

b_4 = {9,10,11,12,13,14,15,16}

4.3. Huffman Code Based Algorithm

An example of the first class of algorithms is one based on the Huffman code. The Huffman code minimizes the code word length of a keyword based on the keyword's probability of occurrence [14,15]. In this scheme, each data item (which represents a column in main memory) corresponding to a label in the alphabet A is listed with its probability of occurrence, and organized into a Huffman tree. Then, memory assignment and duplication are determined by the number of banks and the depth of the tree and its sub branches. Here is an example (see figure 14):

l=4,

R=4,

P(1) = 0.6, P(2)=0.2, P(3)=P(4)=0.1

b_1 = 1; b_2 = 1; b_3 = 2; b_4 = 3,4;

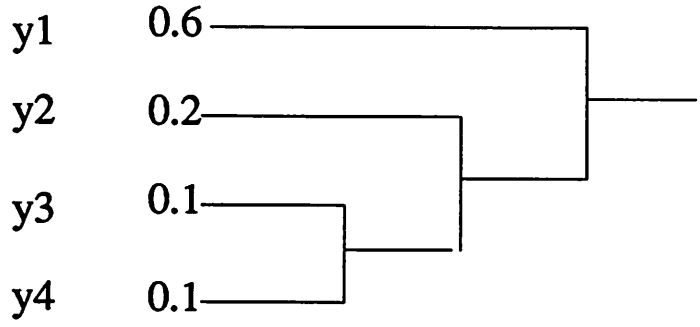

Figure 14 (tree). Data Allocation Based on a Huffman Tree.

Note that there is a nice definitive scheme for duplication, and only simple a priori probabilities are required. However, now the memory banks are used in a very disproportionate way. Bank 1 and bank 2 are accessed 30 percent of the time, while banks 3 and 4 are each accessed only 20 percent of the time. Also, if 4 occurs after 3 very often, putting these two in different banks would significantly improve speedup. This method does not recognize this opportunity at all.

Therefore, in this method, the first order probabilities of the data columns, the number of banks, and the processor to memory speed ratio are specified, and the algorithm determines the duplication and partitioning of the columns. The banks in this scheme may not necessarily be of equal size.

4.4. Sequential Access Algorithm

In the sequential access algorithm, allocation is optimized for sequential access of the R memory banks. If R is less than or equal to P, it can be shown that periodic access of the banks results in the least number of wait states (See section 3.) Periodic access is best suited to a deterministic process, where the sequence of data to be accessed is predetermined. An example of such a process is a Markov process in which the states are the data columns to be accessed, and whose transition matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & . & . & 0 \\ 0 & 0 & 1 & 0 & . & . & 0 \\ 0 & 0 & 0 & 1 & . & . & 0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & 1 \\ 1 & 0 & 0 & 0 & . & . & 0 \end{bmatrix} \qquad (6)()$$

A process which is not deterministic, but whose behavior is not completely random, might be described by the following transition matrix:

$$\begin{bmatrix} 0.1 & 0.7 & 0.05 & 0.03 & . & . & 0.1 \\ 0.05 & 0.1 & 0.65 & 0.02 & . & . & 0.05 \\ 0.05 & 0.1 & 0.1 & 0.8 & . & . & 0.02 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & 0.72 \\ 0.7 & 0.1 & 0.1 & 0.1 & . & . & 0.01 \end{bmatrix} \qquad (7)()$$

In the first (deterministic case), the best allocation scheme is simply to put the necessary data columns in one bank after the other, starting over when there are no more banks. In doing so, we implicitly take advantage of the fact that the probability of state i+1 following state i is 1. In the sequential access algorithm, we take advantage of similar characteristics of the second (non-deterministic) case. The algorithm assumes a fixed number of banks, a fixed bank size, and a training sequence from which first and second order probabilities have been estimated.

The algorithm places data as follows:

i) Set $M_{i,j} = 0$ for all i, j. j=1.

ii) Insert an arbitrary item with label i (which is in A) in bank j. $M_{i,j} = 1$.

iii) For each label k in A, evaluate $\sum_{i=1}^{I} M_{i,j} P$ (i is followed by k). The item k_{max} which maximizes this sum goes into bank (j+1) mod R.

iv) j = (j+1) mod R. $M_{k_{max},j} = 1$.

v) If banks are not full yet, go to step (iii).

vi) Check for unallocated columns. Take the allocated item in the first bank which has the lowest probability of occurrence. If this item is allocated in more than one place, replace

it with the first unallocated item. Else, repeat check for allocated item with next lowest probability of occurrence. Repeat until unallocated item i is allocated, or if no item in current bank can be replaced. $M_{i,j}$=1, where j is bank holding newly allocated item.

vii) If there are still unallocated columns, go to next bank and repeat step (vi).

As you can see, this algorithm optimizes allocation for those processes which have nearly deterministic patterns. Even if the transition matrix does not have the specified form, it may be possible to obtain this form by properly permutating the alphabet.

The sequential access algorithm relies on both first order and second order probabilities. Recall that knowledge of these probabilities are not assumed, and they must be estimated from the training sequence. For large alphabet sizes, good estimation of second order probabilities will require a very long training sequence. This is one disadvantage of this method.

In addition, if the transition matrix of the process does not have a clear maximum in each row, and if it cannot be changed to the desired form, this method may not be suitable, since the situation which motivated its development will no longer hold. However, if the process does have the desired form, this method would result in considerable speedup and require only very simple addressing schemes.

Therefore, in this method, the number of banks, the duplication, and the processor to memory speed ratio are predefined, and the algorithm determines the distribution of the columns. All of the banks will have the same size.

4.5. Score Based Algorithm

The general goal of proper allocation is to minimize contention. This suggests the simple approach of defining the event E_{ij} as the event that label j occurred in P increments or less after label i in the training sequence. Then define a "score" for this event which gives an indication of how likely it is to happen. Such a score could be a simple count (or weighted count)

of these events occurring in the training sequence. Then, an attempt is made to place data columns in pairs with the highest scores in different memory banks. The algorithm is given below:

i) Create a stack of all the labels in A.

ii) Take the item whose label is at the top of the stack, and update the stack. For each bank, evaluate the maximum score that would result between any two member columns of the bank if the new item were to be added. Call this score BankScore (j), where $1 \leq j \leq R$.

iii) Place the new item in the bank j which corresponds to the lowest BankScore.

iv) Repeat steps ii) and iii) until all columns corresponding to the labels in the stack have been allocated.

This method directly attacks the problem of memory bank contention, and is not dependent on exact values for the second order probabilities. (It does not even look at first order probabilities.) However, it has no scheme for duplication, that is, it only partitions the data columns. Duplication can be achieved by simply replicating the scheme obtained from this method an integer number of times. Therefore, if a duplication d and a number of banks R are desired, one can start this algorithm assuming int (R/d) banks, and then duplicating the results d times.

Therefore, this method needs a predetermined number of banks, processor to memory speed ratio, and (integer) duplication, and results in a partitioning scheme, in which the memory banks may not necessarily have the same size.

4.6. Training Algorithm

In this method, allocation is done directly from the training sequence. This is in contrast to the previous methods where first, certain parameters (e.g. probabilities or scores) had to be estimated from the training sequence, and allocation was then based on those parameters. The

allocation method is as follows:

i) Determine a maximum wait, i.e. a number of cycles beyond which we would not like to wait between any two data accesses.

ii) Initialize the available times of all banks to 0. Initialize first cycle.

iii) Receive the current label from the training sequence. Determine the earliest bank available from which the corresponding item can be read (if it has already been allocated), and the earliest bank to which the item can be written.

iv) If the item has not yet been allocated, write it to the next available write location. The next available time for this location is max(current time, ready time(location)) + P.

v) If the item has been allocated{

If the next available read location is ready longer than the maximum wait from now, and if the next available write location is ready earlier than the next available read location{

write the item to the next available write location. The next available time for this location is updated to max (current time, ready time (location)) + P. }

Else{

The next available time for the next available read location is updated to max (current time, ready time (location)) + P. }

}

vi) Increment cycles by 1. If there are still labels in the training sequence, go to step iii).

This algorithm does not directly use first or second order probabilities, but implicitly takes the data accessing statistics into account, provided the training sequence is long enough. It precisely defines the amount of duplication and the partitioning of data in the interleaved

banks. Like the score based algorithm, it directly addresses memory contention at every instant in time.

Therefore, this algorithm presumes no first or second order statistics. It only requires the training sequence, number of banks, and processor to memory speed ratio, and determines the duplication and distribution of the data columns. The resulting scheme does not necessarily have banks of the same size.

It can be seen that both classes of algorithms have their strengths and weaknesses. It is possible to complement one class with another, e.g. use the first class for duplication and the second class for partitioning. In other words, one can specify a new, third class of algorithms which takes into account both the time average and the instantaneous behavior of the statistics within the same algorithm.

5. RESULTS

In this section, we present the results of tests made on the algorithms presented in section 4 using artificially generated data. We compare and contrast the algorithms with the help of these results. In most cases, we generated strings of acoustic labels based on Markov models with synthetically generated transition probabilities (which were normalized to meet the proper criteria.) This procedure seemed reasonable since in most modern speech recognition systems, the English language is modeled by Hidden Markov Models. We wrote programs to generate a Markov transition matrix, and to generate a stream of acoustic labels using a random number generator and the Markov matrix. The alphabet size was 128, and the label sequence was typically 20,000 labels long. We then wrote a program to simulate access of memory banks. This program does not assume any particular order of bank access. The locations of each item are read and stored in a two dimensional array. Labels are then read from a data sequence. If any of the locations of the requested item is not busy, no wait states

are needed and the next item is read. Otherwise, the item must be waited for, slowing the process down. This process is continued until the end of the sequence is reached. The program outputs the number of cycles, the number of cycles spent waiting, the overall speedup, the duplication, and the maximum wait period detected. The method of obtaining test results is shown in figure 15.

The figure of merit by which all the methods were evaluated is the overall speedup. The speedup was measured while changing various system parameters, such as the number of banks R, duplication d, and processor to memory speed ratio P. It must be remembered that in all cases except when cache is used, an upper bound for speedup is min(R,P). Also, the minimum speedup we expect from any data allocation is d, since duplication of main memory d times guarantees a speedup of roughly d. These upper and lower bounds are plotted with some of the graphs for easier comparison.

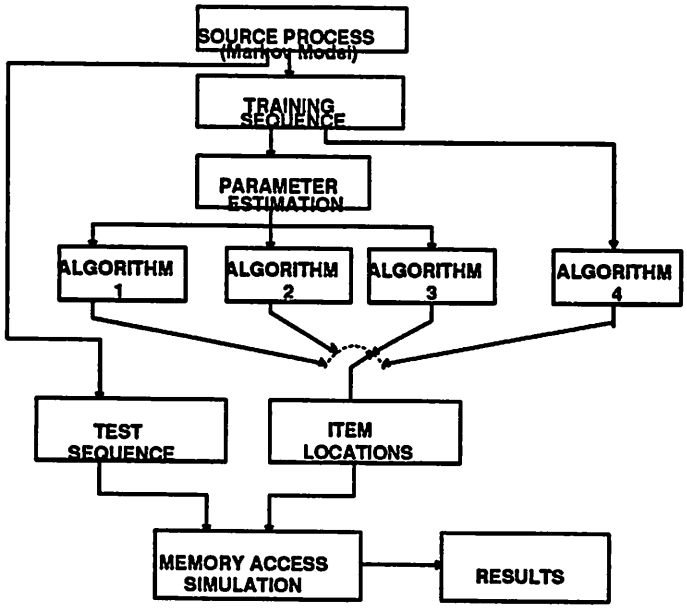

Figure 15 (procedure). Method for Obtaining Test Results.

The speedup is also a function of the type of process generating the sequence of labels. Therefore, the nature of the process used to obtain the test results must be kept in mind, namely that it is first order Markov. To gain some insight into this effect, we generated label sequences from two different Markov processes. The first process has random transition probabilities normalized to meet the necessary conditions for a Markov transition matrix. The second Markov process has a transition matrix of the form shown in equations 6 and 7 in section 4.4. The transition probabilities were first made to decay exponentially with a factor of 0.1 from the maximum value in each row of the transition matrix, and were then normalized to meet the necessary conditions of a Markov transition matrix. Naturally, more is to be gained with the second process, since we can take advantage of the fact that certain label sequences are more likely to occur than others.

Test results for the second method, which optimizes for sequential access, are not shown for the first process. This is because the sequential access method was not intended for processes like the first one, and therefore does not result in significant performance gains.

5.1. First Process

The first process is a Markov process with rather evenly distributed transition probabilities. The transition matrix for this process was created by generating random numbers, and normalizing the random numbers in each row of the transition matrix such that their sum is unity. In this subsection, all algorithms described previously except the sequential access algorithm are tested using a label sequence created using this process. For better comparison, the generic algorithm was also tested (See section 4.2.)

5.1.1. Generic Algorithm

For comparative purposes, the generic algorithm's performance was tested. This algorithm fills the banks with successive columns, and starts over when it reaches the end of the

columns. This action is repeated until all the banks are full. We allocated data using this algorithm, and simulated the memory access to obtain speedup plots vs. processor/memory speed ratio, number of banks, and duplication. These results are shown in figure 16, figure 17, and figure 18.

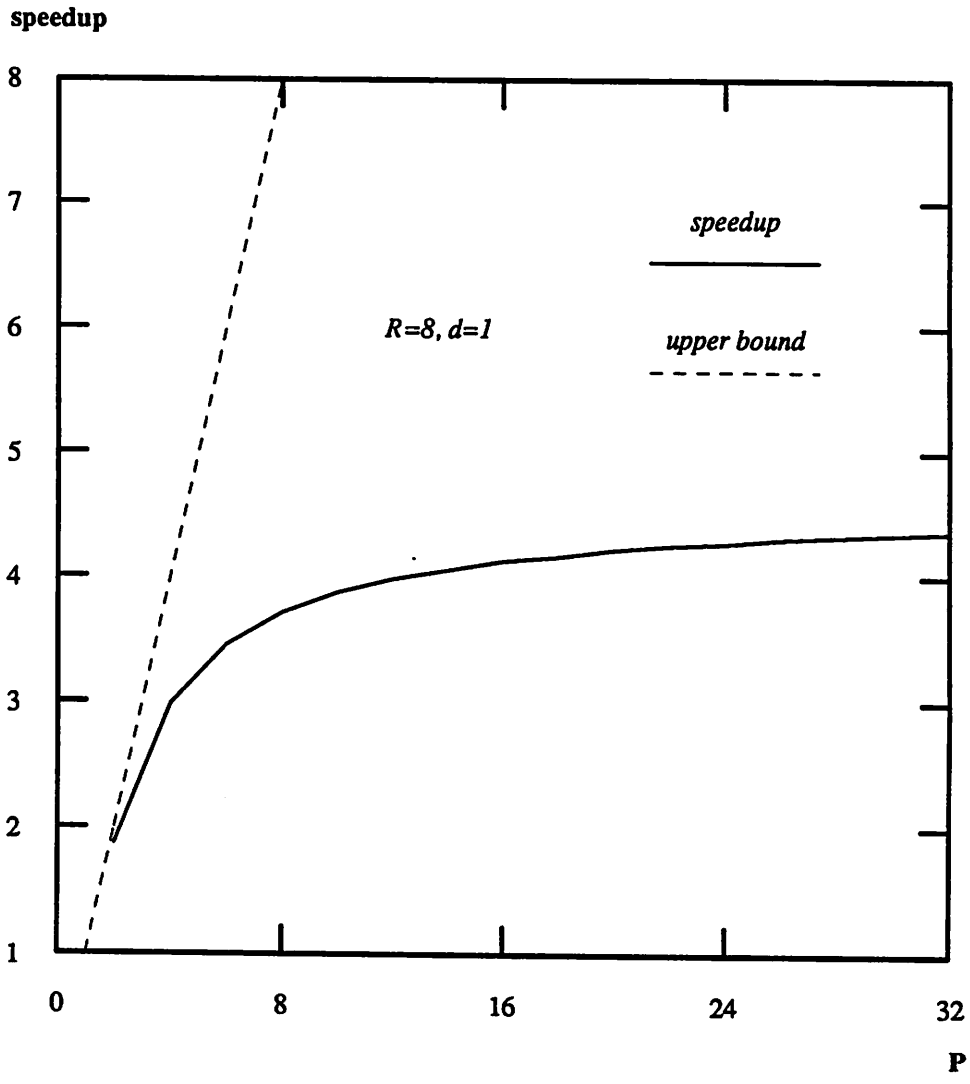

Figure 16 (genericvsP1). Speedup as a Function of Processor/Memory Speed Ratio for the First Process (Generic Algorithm)

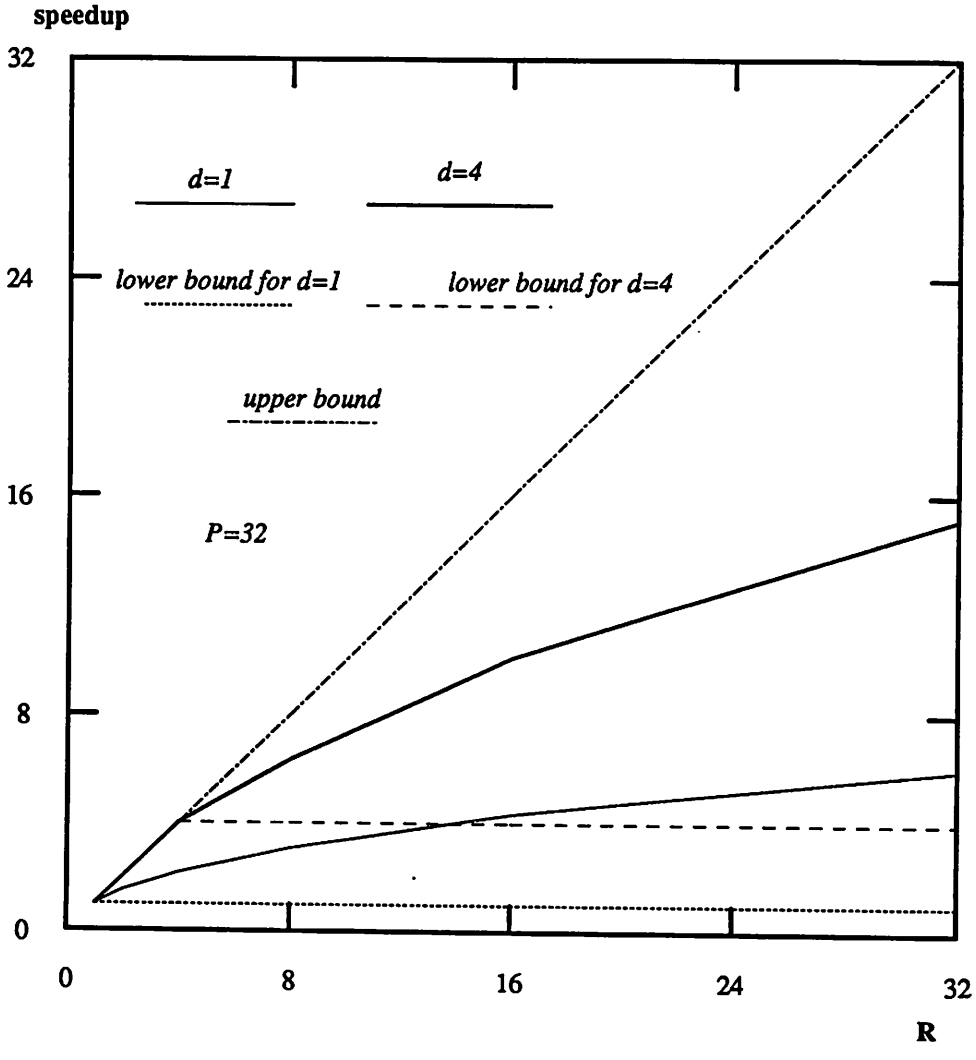

Figure 17 (genericvsR1). Speedup as a Function of Number of Banks for the First Process (Generic Algorithm)

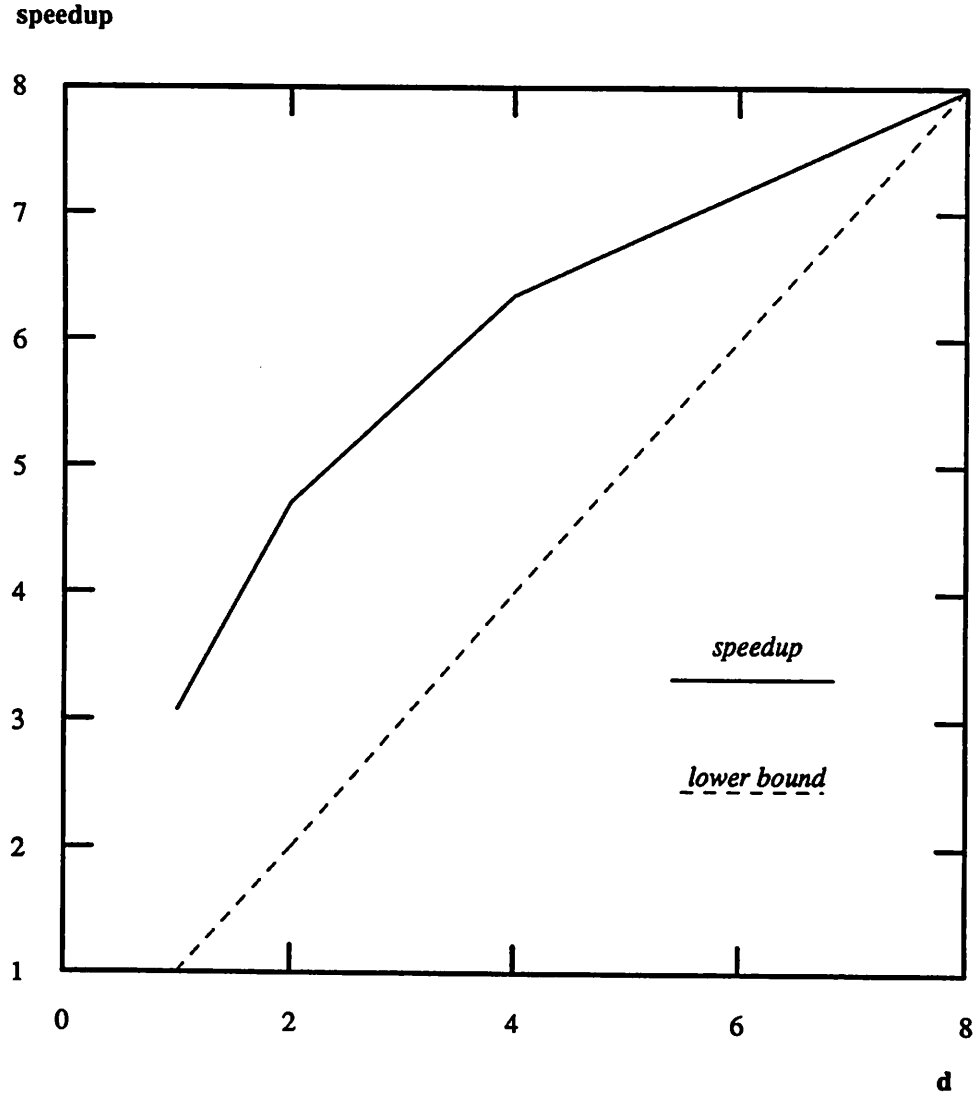

Figure 18 (genericvsd1). Speedup as a Function of Duplication for the First Process (Generic Algorithm)

The speedup seems to be a function of P only below certain values of P. This is not surprising since at lower values of P, less speedup is possible. Beyond a certain P, however, memory contention becomes the main factor in speedup, and having larger values of P does not help alleviate the contention bottleneck.

5.1.2. Huffman Code Based Algorithm

The next algorithm is the Huffman code based algorithm. For a full description of this algorithm, please see section 4.3. To evaluate the performance of this algorithm, we recorded the resulting speedup vs. processor to memory speed ratio and number of banks. The results are shown in figure 19 and figure 20. Speedup increases with both of those quantities just as one would expect. As mentioned before, this method does not pay attention to temporal conflicts. Therefore, two columns might have low probability of occurrence overall, and therefore get grouped into one bank. If these two columns occur within P cycles of each other, however, they will create contention. If the probability of these two particular columns is low enough, this does not become much of a factor. However, if a bank happens to be full of columns which have temporal conflicts, the contention is significant even if the probability of occurrence of each of the columns is low. In the case of our test, the random process generating the stream of labels is a first order Markov process with rather evenly spread transition probabilities, and the temporal conflicts mentioned above are not very significant. That is, there is not much to be gained from observing the higher order statistics to separate columns with high probability of following one another.

Once again, the effect of P on the speedup is felt only at lower values of P, after which speedup is fairly independent of the processor to memory speed ratio.

This algorithm not only shows no improvement, but in fact is inferior when compared to the generic algorithm. This is not surprising considering the nature of the process. Intuitively, every time a label arrives, little can be predicted about which label comes next, leaving little to exploit during allocation. In addition, the generic algorithm is a more "neutral" method in terms of allocation. Departing from that allocation could either improve or deteriorate performance. In short, the performance gain for such a process is very limited regardless of the allocation we use.

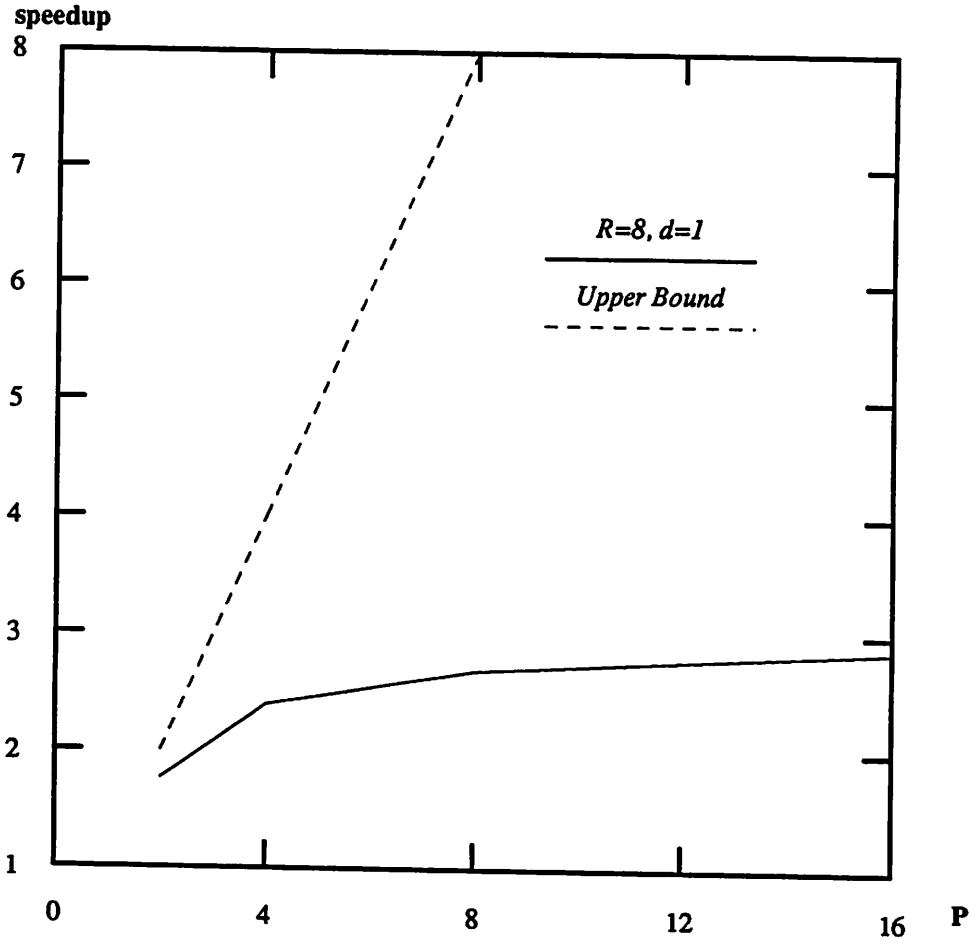

Figure 19 (huffvsP). Speedup as a Function of Processor/Memory Speed Ratio (Huffman Code Based Algorithm)

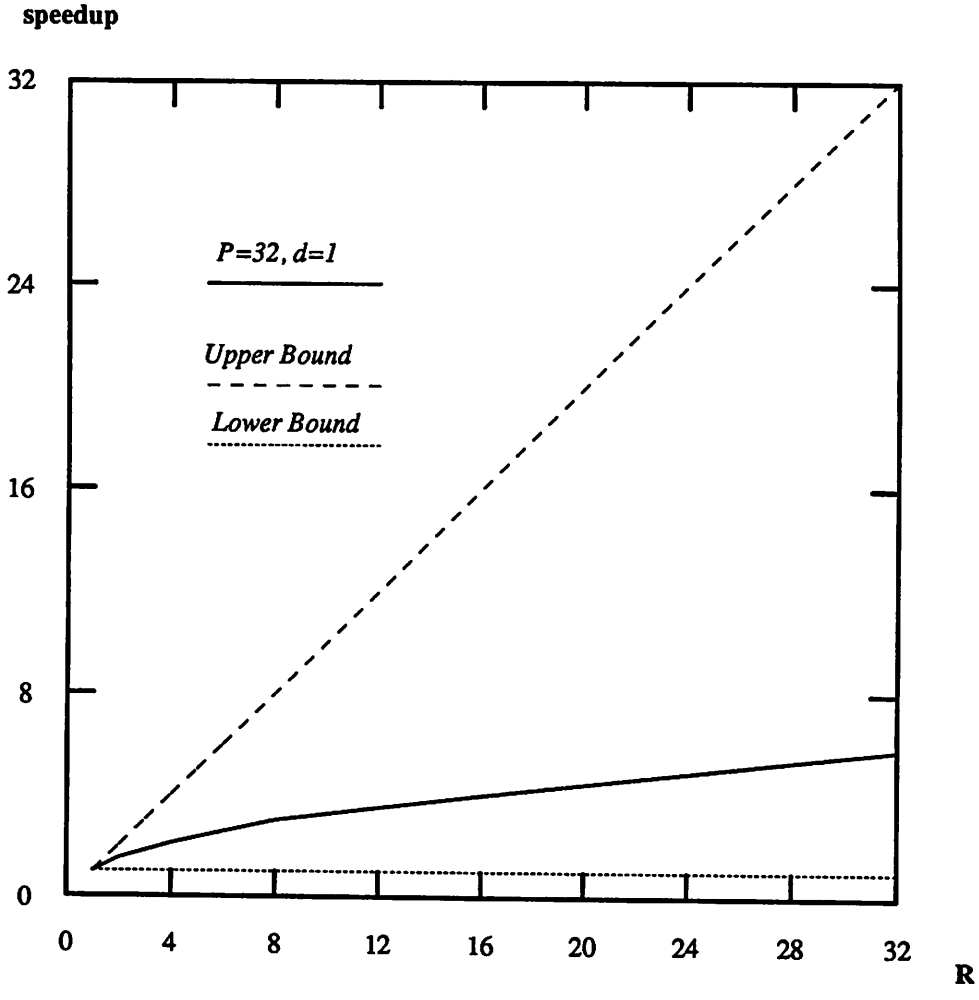

Figure 20 (huffvsR). Speedup as a Function of Number of Banks for the First Process (Huffman Code Based Algorithm)

5.1.3. Score Based Algorithm

The third algorithm is score based, and is completely described in section 4.5. This algorithm produces solutions with integer duplication. We wrote a program to implement this algorithm, and recorded the speedup resulting from this method. The plots in figure 21, figure 22, figure 23, and figure 24 show the variation of speedup with processor to memory speed ratio, number of banks, and duplication. From these plots it is evident that the number of banks has a large effect on the speedup. The score used in this measurement was a simple count. That is, the number of occurrences for label i within P time increments of label j is

counted. A weighted score may also be used, in which larger weight factors are assigned to occurrences which are closer together. Surprisingly, using such weighting did not improve the speedup significantly, if at all.

In this algorithm as well, speedup is affected by changes in P only for lower values of P After a certain point, P does not affect the speedup. To illustrate this, the plot in figure 24 shows that doubling R has a greater effect on speedup than quadrupling P.

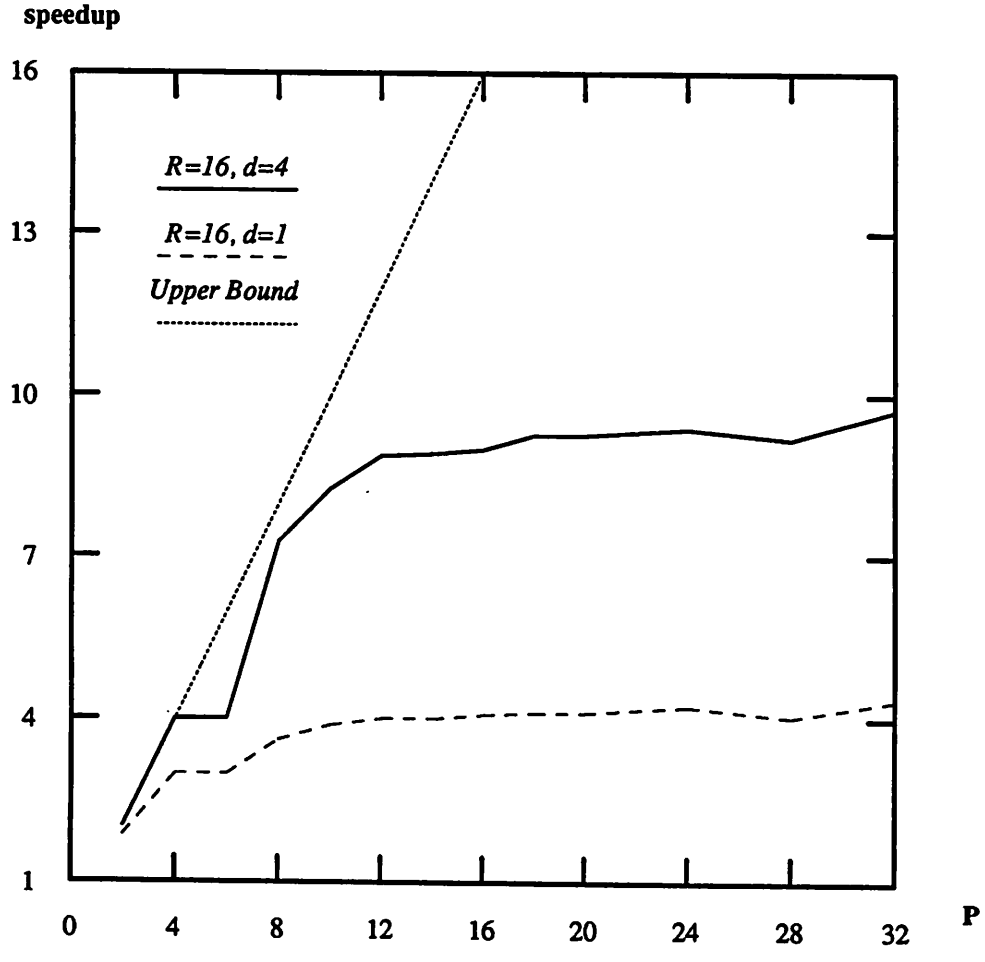

Figure 21 (scorevsP1). Speedup as a Function of Processor/Memory Speed Ratio for the First Process(Score Based Algorithm, R=16)

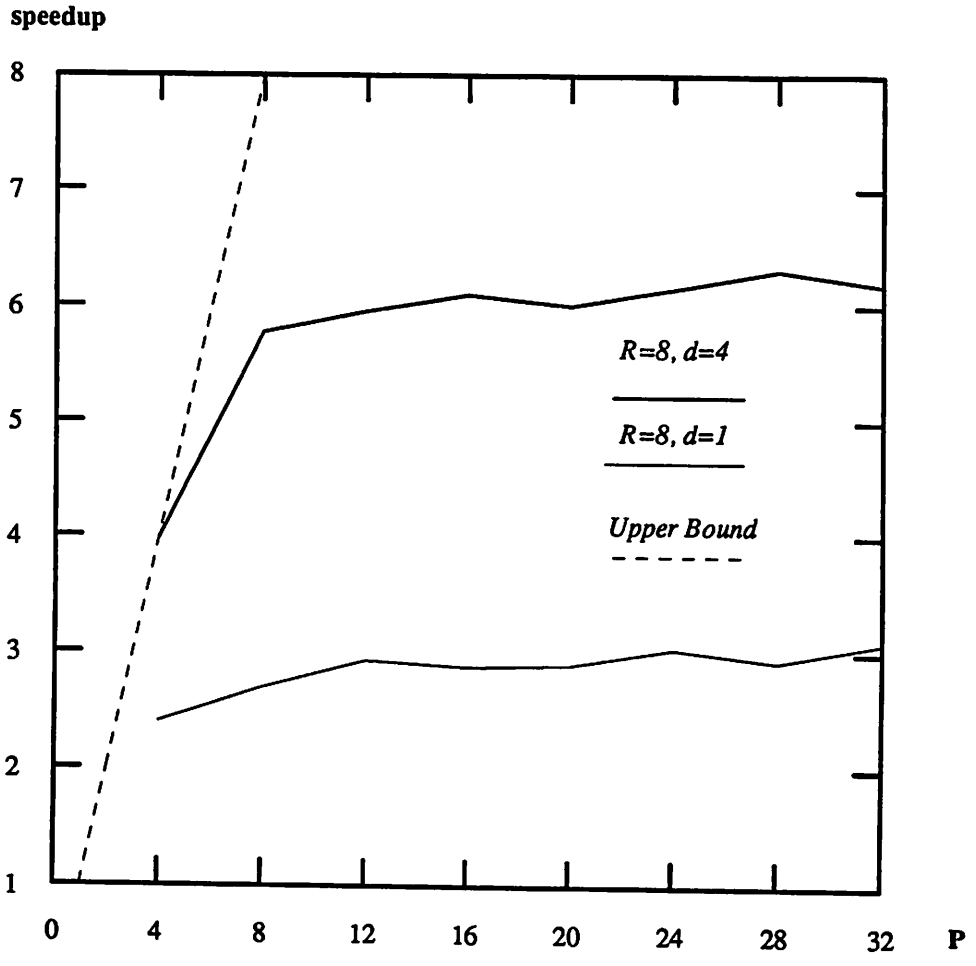

Figure 22 (scorevsP8). Speedup as a Function of Processor/Memory Speed Ratio for the First Process(Score Based Algorithm, R=8)

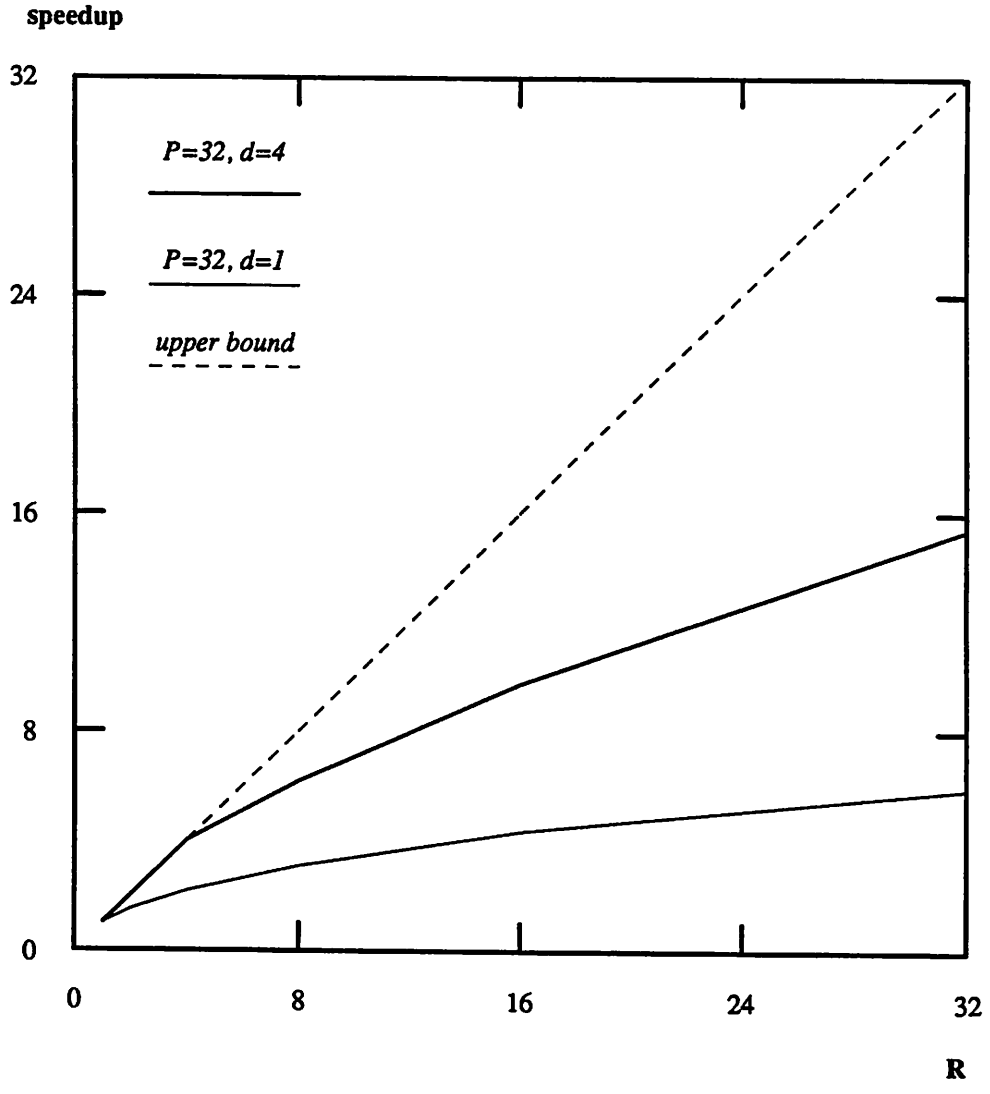

Figure 23 (scorevsR1). Speedup as a Function of Number of Banks for the First Process (Score Based Algorithm)

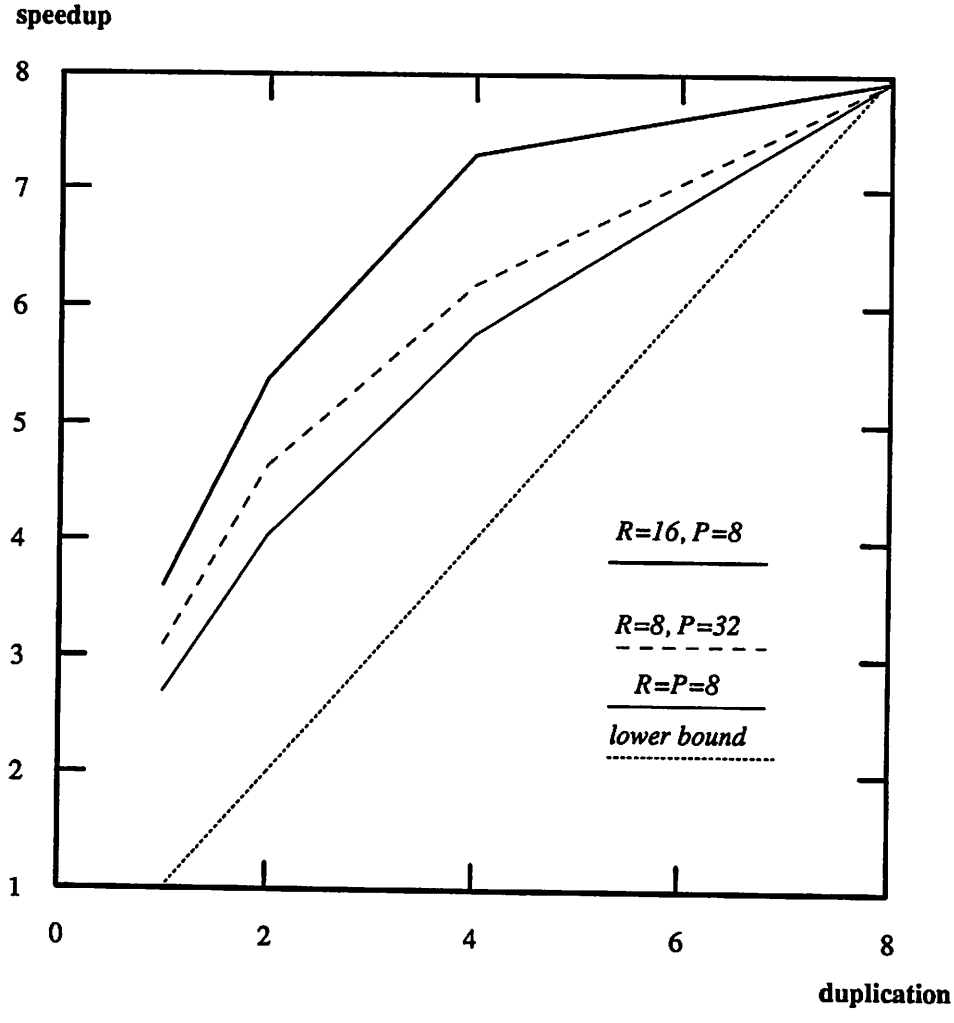

Figure 24 (scorevsd1). Speedup as a Function of Duplication for the First Process (Score Based Algorithm)

Once again, a comparison to the generic algorithm shows little improvement (and sometimes even a degradation) in performance using the score based algorithm. This is attributed to the process generating the labels. There is little inherent information in this process, making it difficult to improve speed by different allocation schemes.

5.1.4. Training Algorithm

The last algorithm is the training method. In this method, no statistics are needed to allocate columns. Items are taken directly from the training sequence and allocated. This

method is described more thoroughly in section 4.6. Once again, the speedup was observed as we changed the number of banks, the duplication, and the processor to memory speed ratio. Unlike the two previous methods, this algorithm determines the duplication. When running the algorithm, the user sets a number of cycles beyond which it is not desirable to wait. This number of cycles is called max wait, and has a significant effect on the allocation. A smaller max wait will demand more duplication, and vice versa.

The plots in figure 26, figure 27, and figure 28 show the variation of speedup with processor to memory speed ratio, number of banks, and duplication. As in the previous algorithms, P has little effect on the speedup after a certain point. The duplication resulting from the algorithm changes with respect to the specified max_wait. These changes are shown in figure 25. This plot shows how max_wait affects duplication after it reaches a certain value. As max_wait approaches P (processor to memory speed ratio), no duplication is required, since the maximum time a bank can be busy is P processor cycles anyway. An interesting trend happens in the duplication dictated by the algorithm when the number of banks R varies. This is shown in figure 29. In this simulation, max_wait was held constant as R was varied. First, duplication demanded was higher, but this duplication decreased above R=16.

It is evident from the plots that there is improvement is some areas. For example, the variation of speedup with duplication is slightly better in the case of the training algorithm. Nonetheless, there is still no tremendous improvement over the generic algorithm.

5.1.5. The Use of Cache

So far, the assumption has been that the use of high performance (and high cost) memory technologies is to be avoided, hence the development of interleaving algorithms to improve the bandwidth of main memory. The result of these methods is improved speed at the cost of some possible duplication, and more complex addressing, rather than the use of

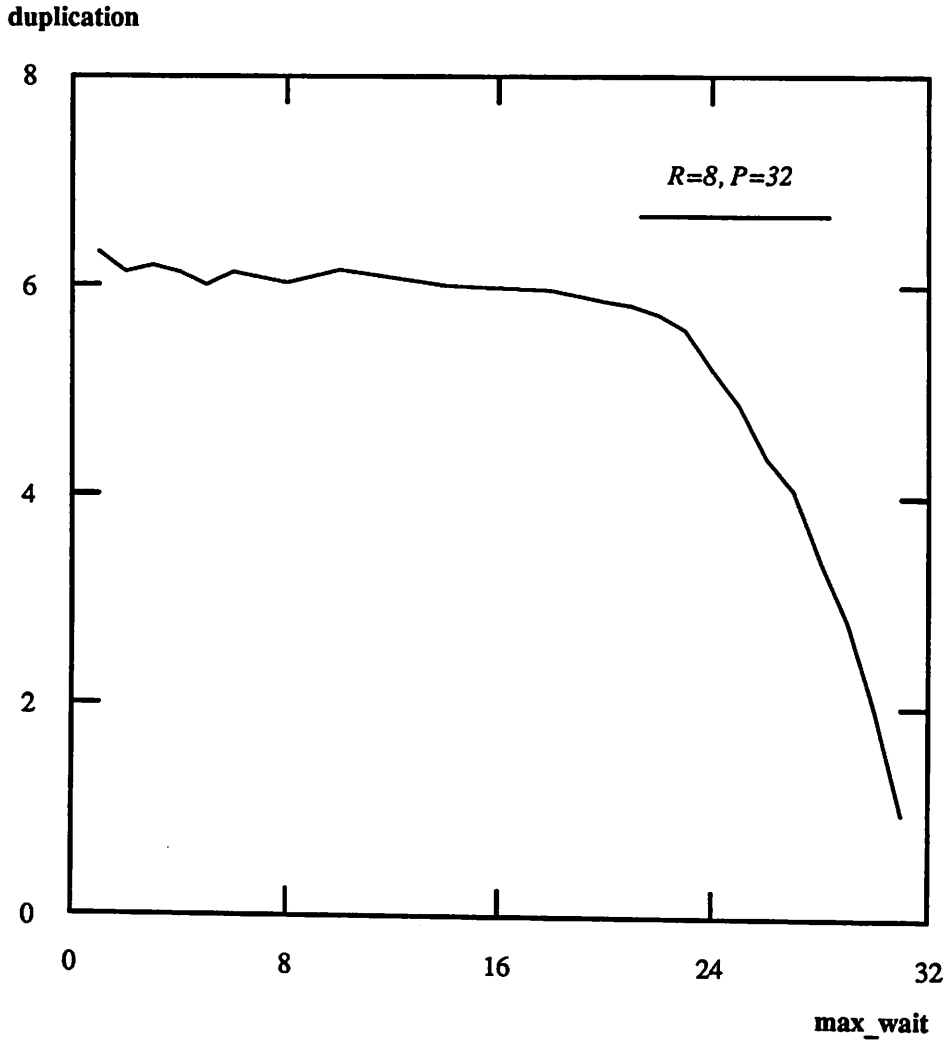

Figure 25 (traindvsmax_wait). Duplication as a Function of the User Specified "max_wait" in the Training Algorithm

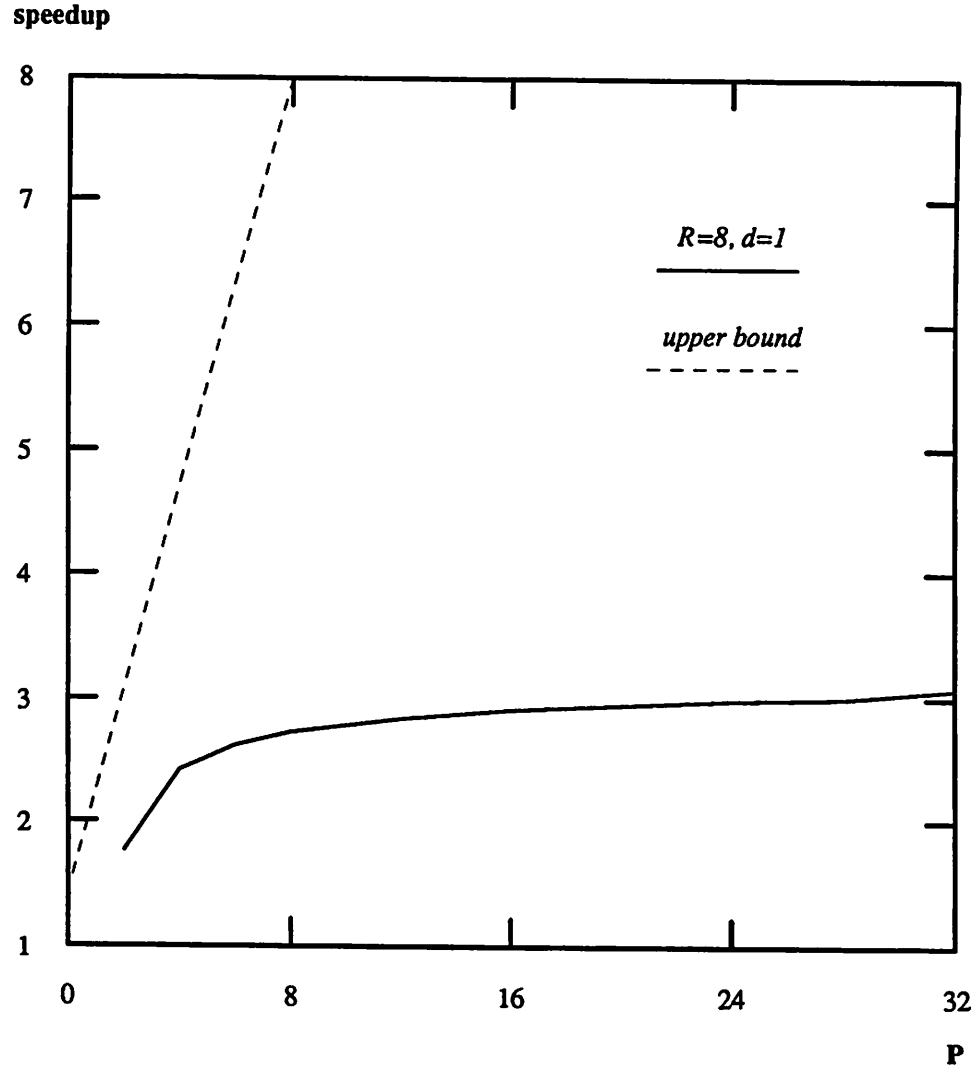

Figure 26 (train1vsP1). Speedup as a Function of Processor/Memory Speed Ratio for the First Process (Training Algorithm)

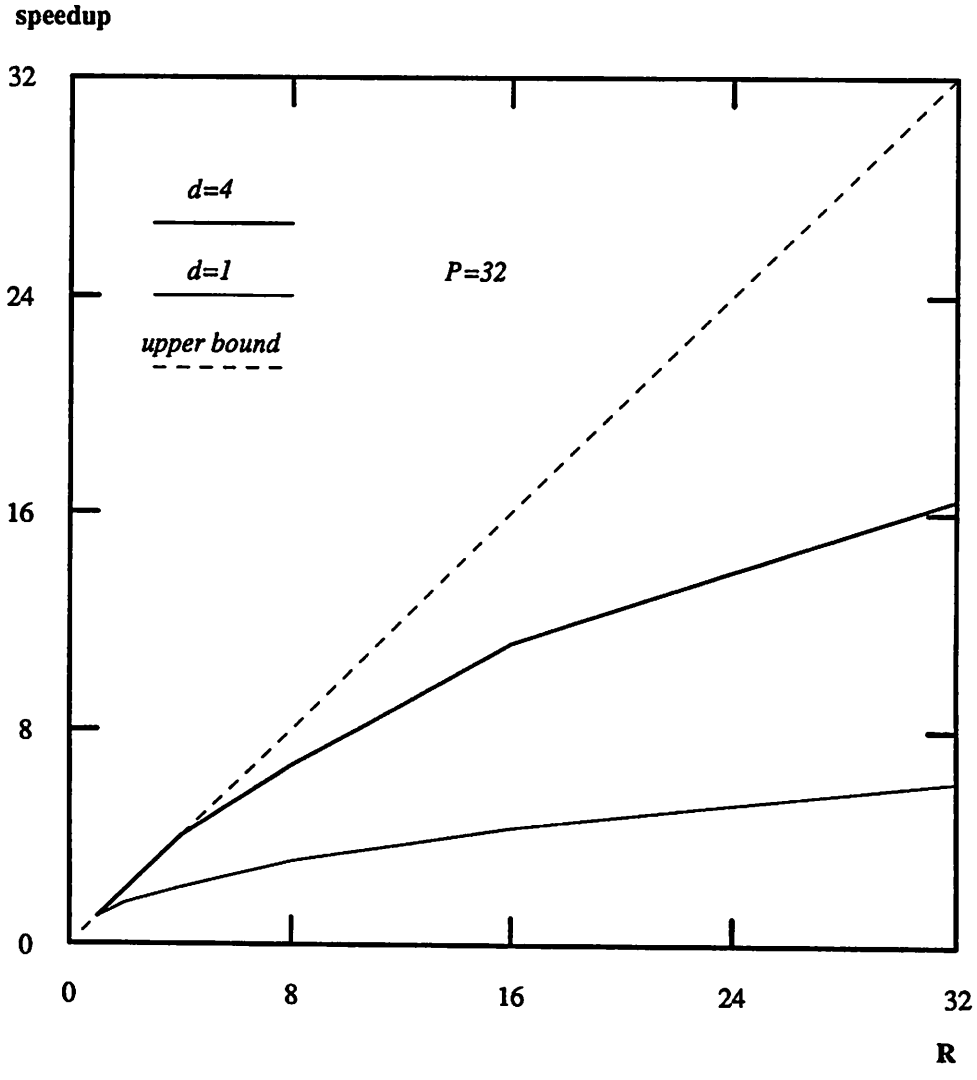

Figure 27 (train1vsR1). Speedup as a Function of Number of Banks for the First Process (Training Algorithm)

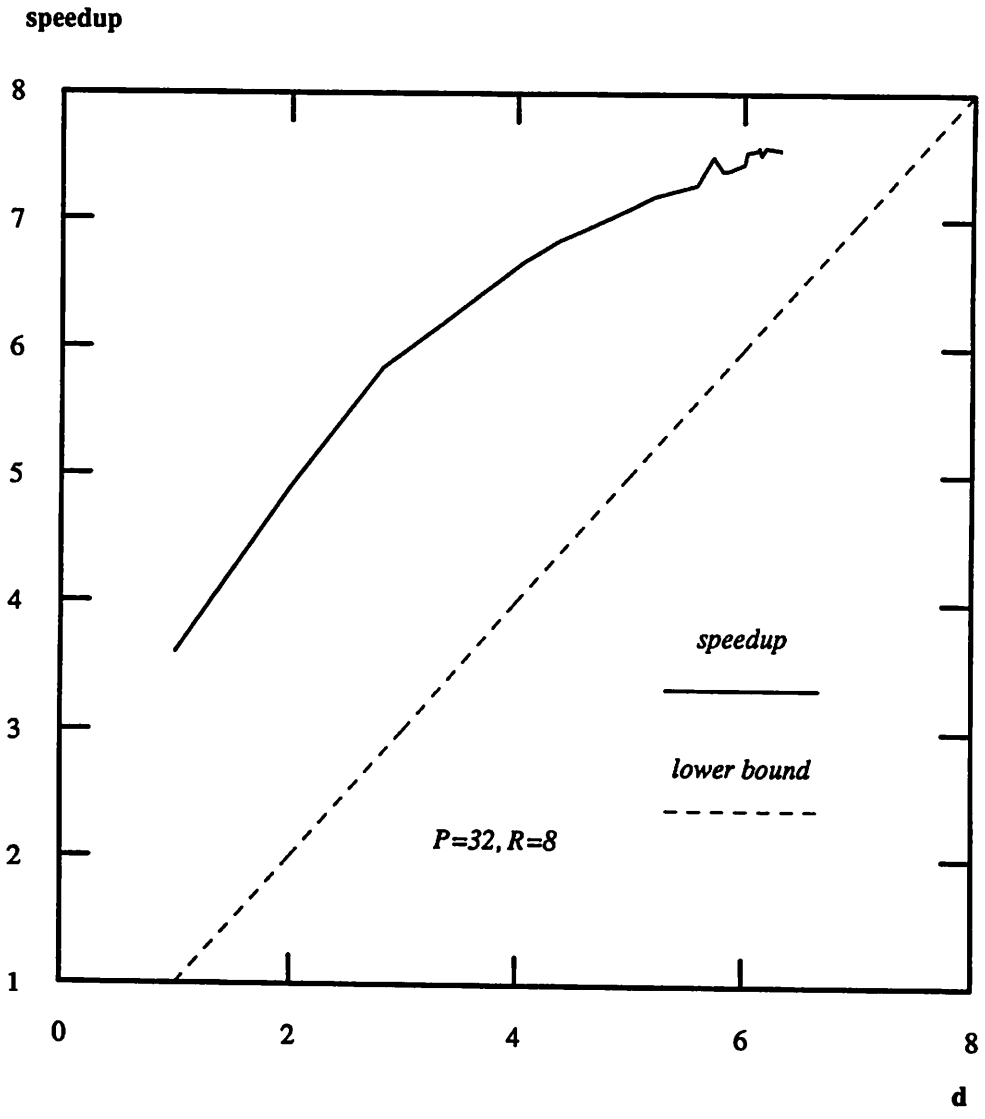

Figure 28 (trainvsd1). Speedup as a Function of Duplication for the First Process (Training Algorithm)

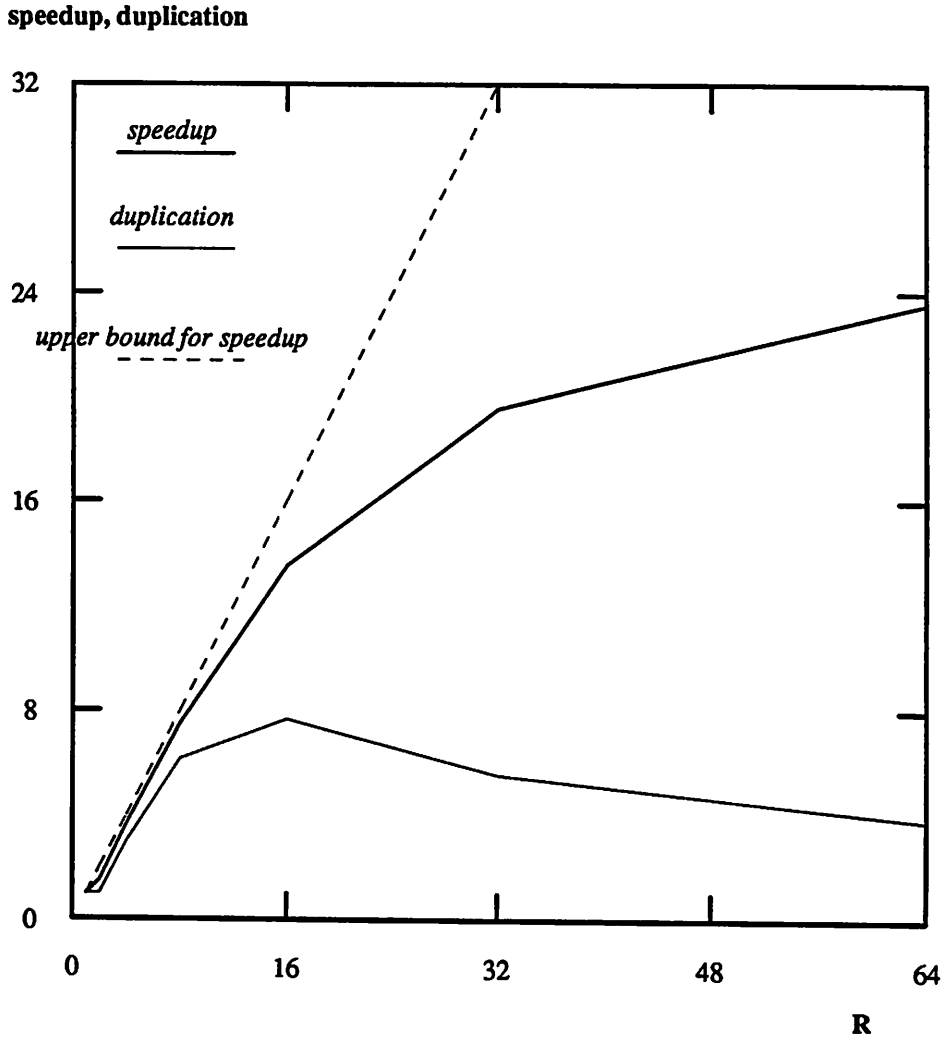

Figure 29 (trainvsR1). Speedup and Duplication as a Function of Number of Banks when max_wait is Held Constant

large amounts of fast memory technologies. However, the tradeoff between technology and memory interleaving is not absolute. That is, smaller amounts of fast technology (cache) can be used in conjunction with memory interleaving to improve overall bandwidth.

In this section, we illustrate the qualitative effects of cache size on speedup. We chose to do this only for the score based and training methods, since they are superior to the others. In these tests, the size of the cache refers to the number of columns the cache can hold. So, if the cache size is equal to the alphabet size, the entire main memory is in effect implemented

as cache. In simulating the cache, a very simple model was used. The cache access time was assumed to be one processor cycle regardless of cache size, and the replacement scheme was chosen as the least recently used (LRU) scheme. So, if the requested item was not in the cache, it would be written over the least recently used item in the cache from main memory. It should be noted that the maximum speedup possible when using cache is not min(R,P), but rather (P*processor cycle time)/cache access time. In our simple model, this quantity is simply P, since the cache access time is equal to the processor cycle time. Also, by not taking the effect of cache size on access time into account, the simulation results in an optimistic figure for speedup, especially for larger cache sizes.

An indication of the effectiveness of the cache is the slope of the curve of speedup vs. cache size. A higher slope shows that the additional amount of cache used results in a larger amount of speedup, making the additional amount of cache worthwhile.

The variation of speedup with cache size is shown in figure 30. The speedups were measured only for cache sizes which are a power of 2. The case where R=1 and d=1 corresponds to no interleaving, that is, an SISD architecture where the only speedup attained is through the use of cache. As you can see, the speedup gained by using cache drops dramatically as the cache size is reduced from that of the whole memory (using fast technology for the whole memory.) Also, the slope of the curve is low at small cache sizes, and the big improvement comes when cache is large. This defeats the purpose of having cache in the first place, which is that a small amount of cache could result in substantial speedup. It seems that cache is slightly more useful at higher duplications for both methods, but still not very effective. Once again, these results are dependent on the nature of the process. The label sequence used in these experiments was generated by a first order Markov process with evenly distributed transition probabilities. Therefore, there is no bias in favor of using certain labels more often than others, and for this kind of process, it is not surprising that the cache is not very

useful.

The results in this section have shown that for the first Markov process, i.e. one with rather evenly distributed transition probabilities, no one algorithm stands out as achieving much better speedup. In fact, none of the algorithms we have put forward for clever data allocation is any better than a regular generic algorithm. It was also seen that the processor/memory speed ratio affects the speedup when it has smaller values, and has little effect after a certain point. In the next section, we look at the improvement achieved by our

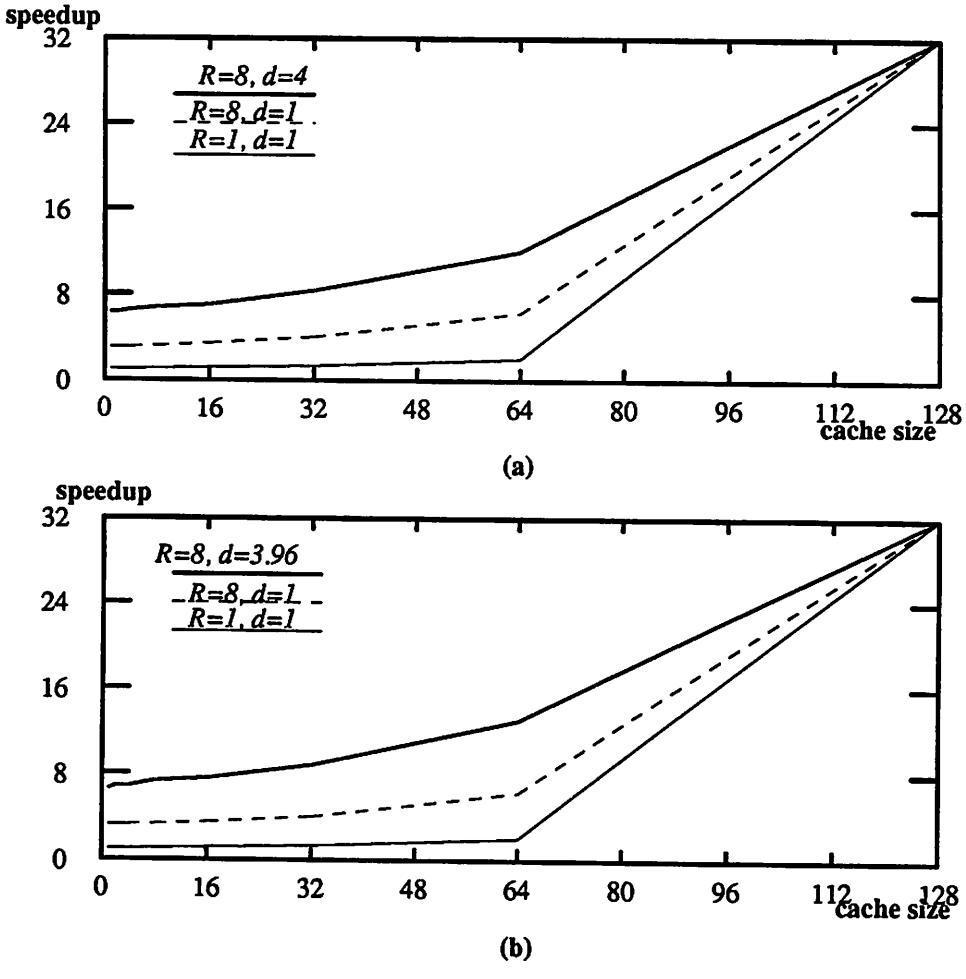

Figure 30 (cache1). Speedup as a Function of Cache Size for the First Process for (a) Score Based Algorithm, and (b) Training Algorithm

algorithm when the process generating the label sequence does not have evenly distributed transition probabilities.

5.2. Second Process

The second process for generating label sequences is also a Markov process, but has the form of equations 6 and 7 in section 4.4. Within each row, the transition probabilities were generated by first putting 1 in the position with the maximum probability (the upper diagonal positions). The other positions decayed exponentially from the maximum position with a decay factor of 0.1. The numbers in each row were then normalized to make the row sum equal to 1. Naturally, in this process, there is more inherent information to take advantage of. Some label sequences will occur more than others, and allocation can be optimized for these sequences. Once again, for better comparison, the generic algorithm was also tested.

The speedup achieved using the Huffman code based algorithm deteriorates in this process because the instantaneous conflicts are more significant, resulting in more wait states on the average. For this reason, the results of the huffman code based method are not shown for the second process.

5.2.1. Generic Algorithm

Once again, the generic algorithm was used for allocation. The resulting configuration was then used in a memory access simulation to obtain plots of speedup versus number of banks, duplication, and processor to memory speed ratio. Note the difference in performance between this case and that of the same algorithm for the first process (figure 16, figure 17, and figure 18) The process now has more more bias towards certain events, and hopefully there is more room for improvement. For this process, there is some predictability as to which label comes next, and taking advantage of this can create substantially better performance.

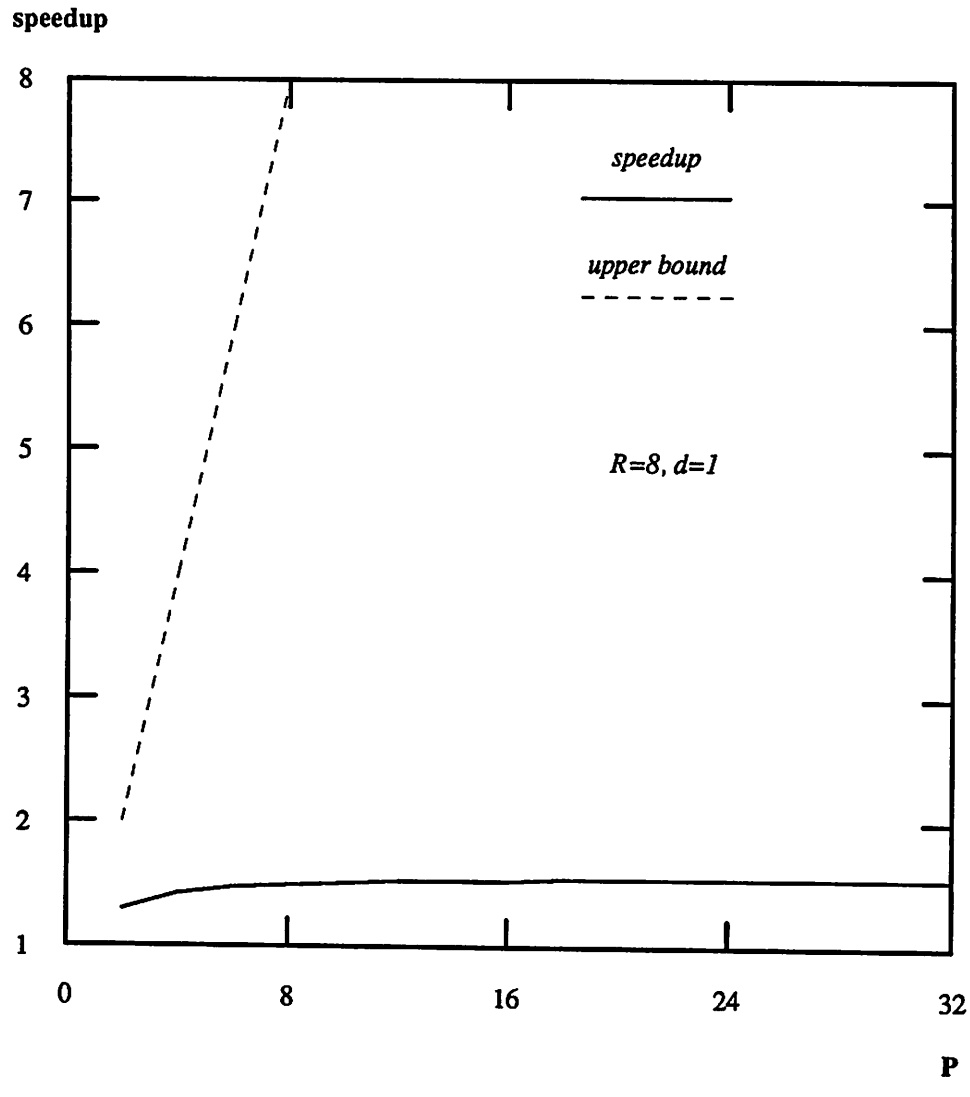

Figure 31 (genericvsP2). Speedup as a Function of Processor/Memory Speed Ratio for the Second Process (Generic Algorithm)

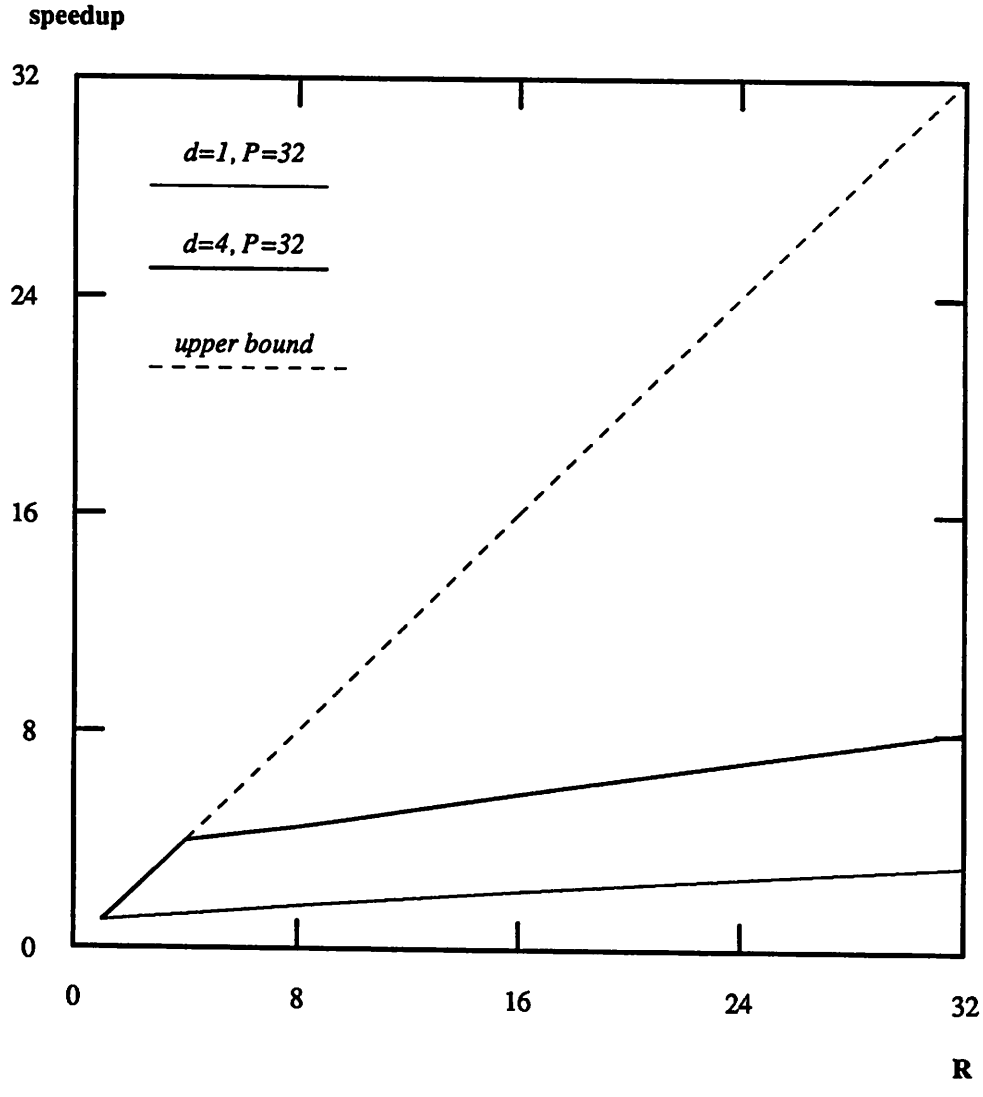

Figure 32 (genericvsR2). Speedup as a Function of Number of Banks for the Second Process (Generic Algorithm)

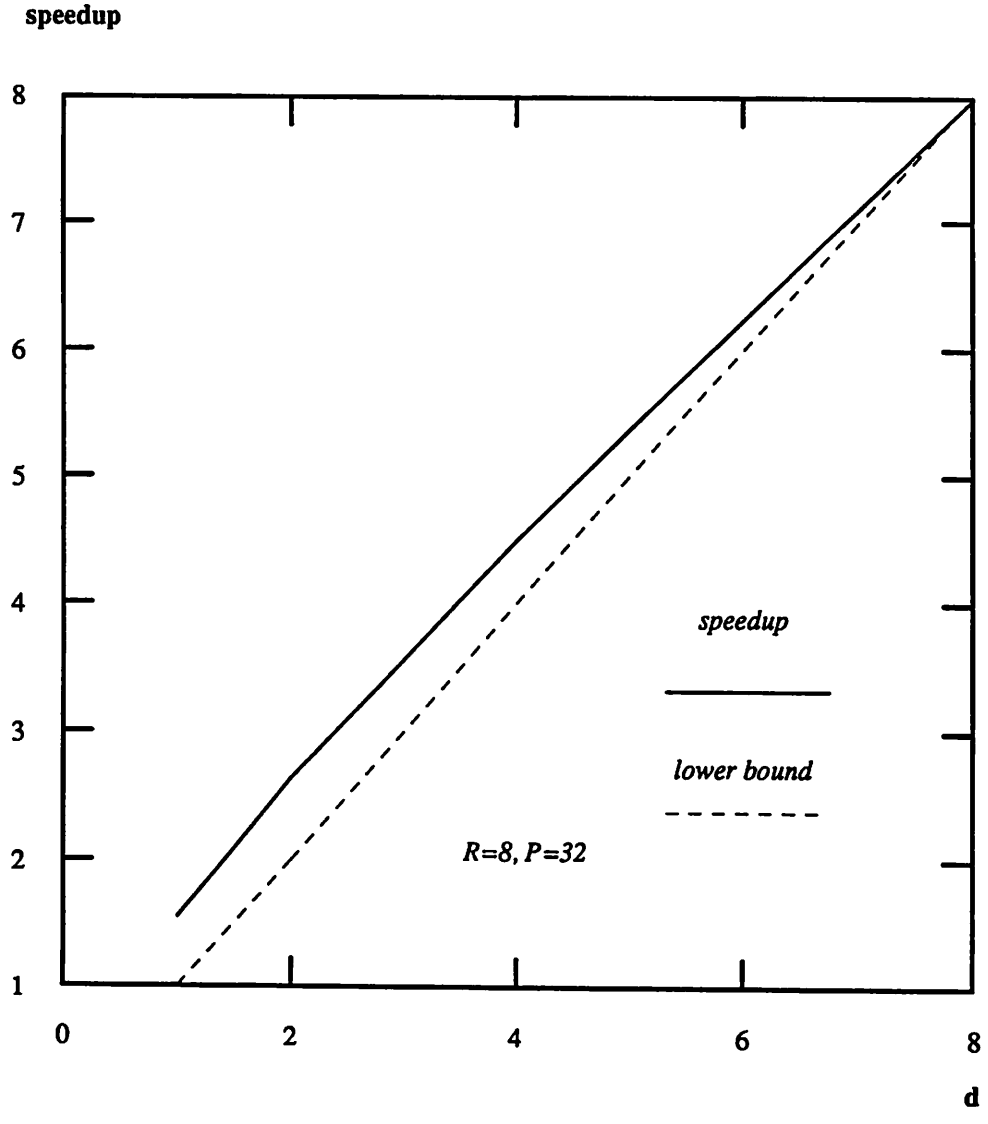

Figure 33 (genericvsd2). Speedup as a Function of Duplication for the Second Process (Generic Algorithm)

5.2.2. Sequential Access Algorithm

The sequential access algorithm optimizes for sequential access. For a full description, please see section 4.4. As described before, this method is suited for processes which are "almost" deterministic, i.e. the sequence of columns to be accessed is almost repetitive. To create such a sequence, the transition matrix of the Markov process had to be modified. In each row of the transition matrix, the transition probabilities decay exponentially as they

occupy further and further positions from the off diagonal elements. The factor of this exponential decay can be easily varied. Naturally, the elements are all normalized to satisfy the necessary criteria for a Markov transition matrix.

Once again, to evaluate the performance of this algorithm, we generated the sequence of columns, allocated the columns in the banks based on the transition probabilities of this stream, and simulated the memory access and recorded the speedup achieved. We recorded the changes in speedup with respect to processor/memory speed ratio, number of banks, and duplication. These results are plotted in figure 34, figure 35, and figure 36. Obviously, a steeper decay in the Markov matrix results in more speedup (The process approaches the deterministic case.)

A major drawback of this scheme is the difficulty in obtaining the necessary statistics from the stream of columns. For a small alphabet size, a training sequence of reasonable length will result in fairly accurate estimation of the transition probabilities. For a large alphabet size, however, the length of the training sequence must grow dramatically. In figure 37, we plotted the average errors resulting from the calculation of transition probabilities from a training sequence. The transition probabilities were calculated by counting the number of times label i would follow label j in the sequence, and dividing by the total number of times that two labels follow each other, i.e. sequence length -1. The error calculation was then made by comparing the calculated transition probabilities to the actual transition probabilities. As you can see, it does not take a very large alphabet size to need a tremendously long training sequence.

The kind of performance seen in this method is considerably better than that of the generic algorithm. Speedups are consistently greater for various cases. This is natural, since the inherent information in the process is being taken advantage of more in the sequential access algorithm. Also, the process happens to be favorable to the sequential access algorithm.

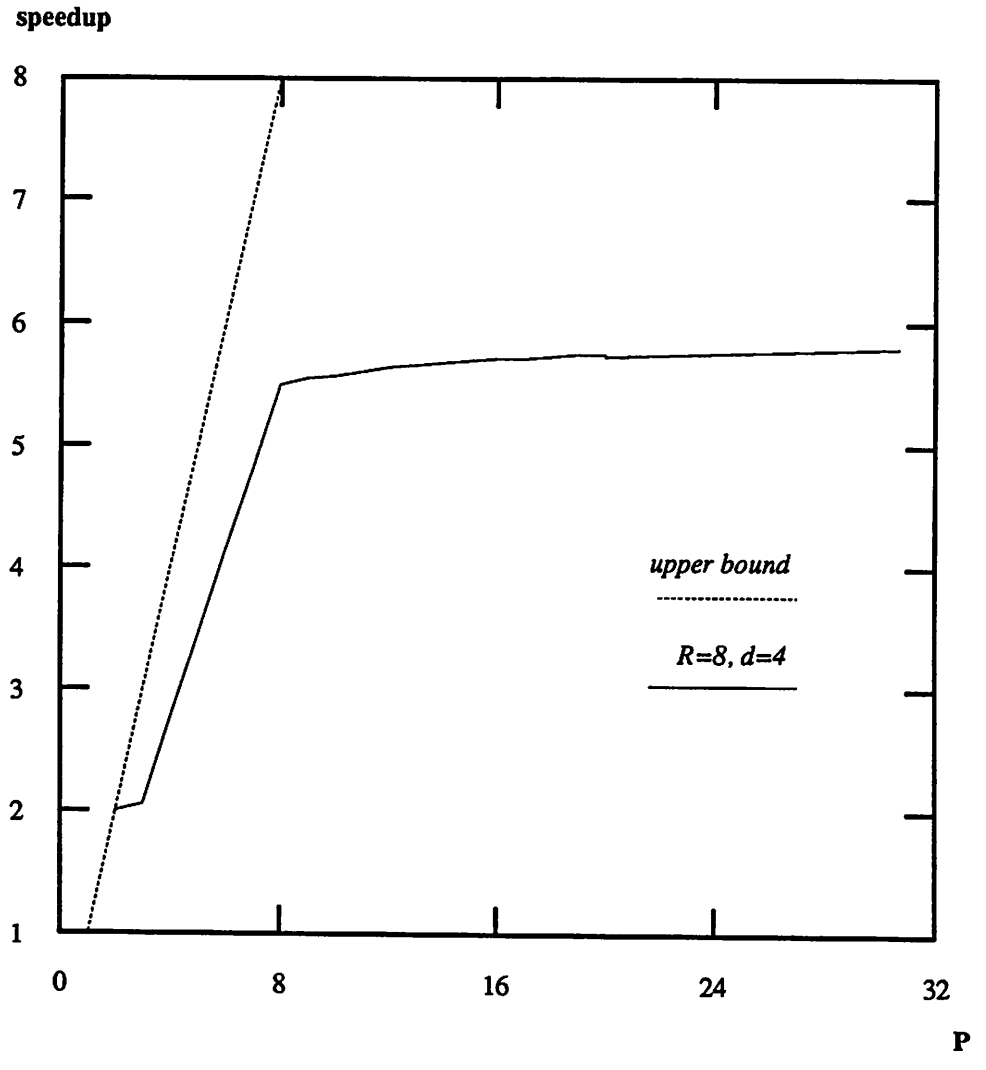

Figure 34 (seqvsP). Speedup as a Function of Processor/Memory Speed Ratio for the Second Process (Sequential Access Algorithm)

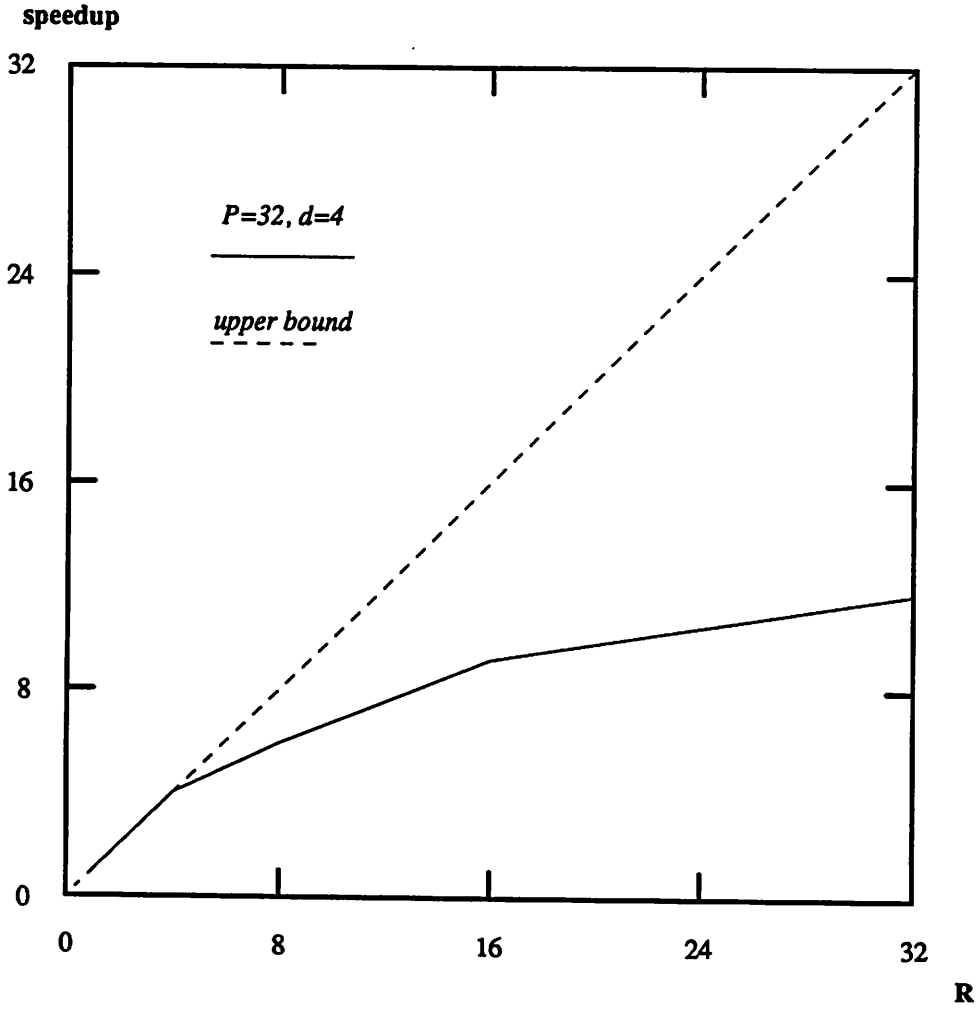

Figure 35 (seqvsR). Speedup as a Function of Number of Banks for the Second Process (Sequential Access Algorithm)

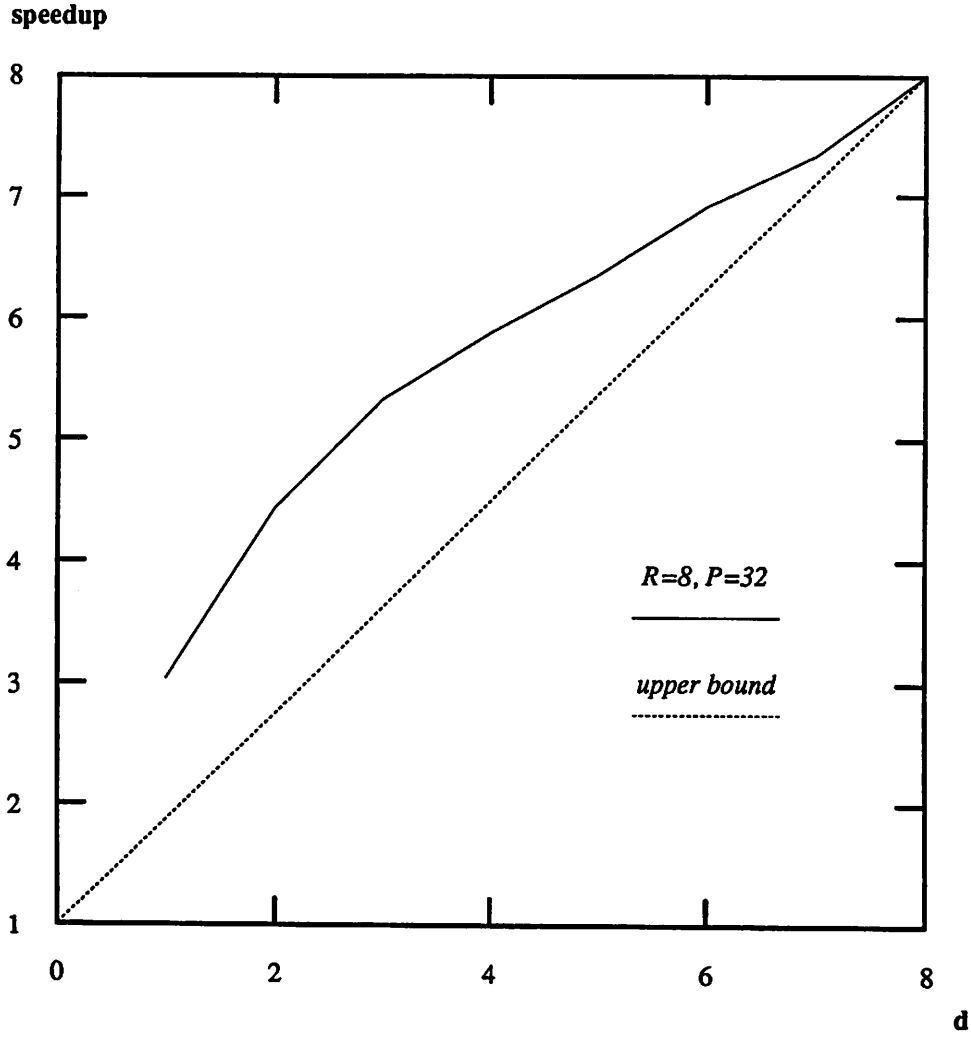

Figure 36 (seqvsd). Speedup as a Function of Duplication for the Second Process (Sequential Access Algorithm)

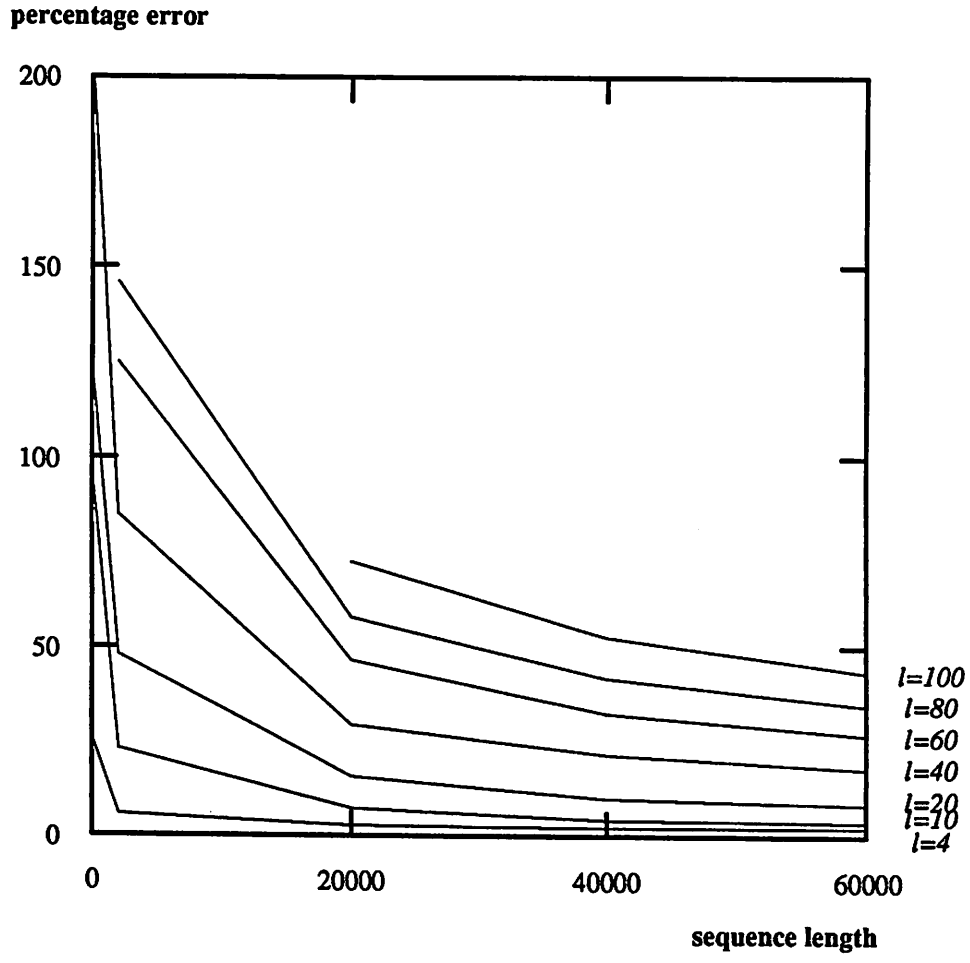

Figure 37 (average_error). Average Percentage Error in Estimation of Transition Probabilities as a Function of Alphabet Size and Sequence Length

Nonetheless, the performance does not even come close to the upper bounds, as seen in figure 35. Therefore, we should look for further improvement. As we have seen before, the effect of P is felt only below certain values.

5.2.3. Score Based Algorithm

The score based algorithm was also used to allocate columns to various banks for the second process. The resulting scheme was used in a memory access simulation, and speedup was plotted for changing processor/memory speed ratio, number of banks, and duplication. (figure 38, figure 39, and figure 40.) Like the previous cases, processor/memory speed ratio

affects the speedup only when it is less than a certain value. Once again, this is illustrated in figure 38 as well as figure 40, where doubling R improves speedup more than quadrupling P.

In contrast to the first process, the speedups achieved by this algorithm are consistently much higher than those using the generic algorithm (figure 31, figure 32, and figure 33.) It should be noted that the scores used in the algorithm were simple, non-weighted counts of labels following other labels within P cycles of each other. Adding weights to the counting process did not improve speedup significantly, if at all. This result reiterates the concept that the nature of the process greatly affects the performance of the allocation algorithm. Where there is more bias and more inherent information, there is more to be gained in terms of performance.

It is also interesting to see the effect of changing the process on the same algorithm. Changing the process does not even nearly effect the score based method (figure 21, figure 38, figure 23, figure 39, figure 24, and figure 40) as much as it does the generic algorithm (figure 16, figure 31, figure 17, figure 32, figure 18, and figure 33) This suggests that the algorithm is adapting the allocation to the process, and is more robust than the generic algorithm.

5.2.4. Training Algorithm

The training algorithm was also used for the second process. The resulting allocation was tested by the memory access simulation, and the results are shown in figure 41, figure 42, and figure 43. From these plots, it is safe to conclude that the training method is in fact consistently much better than the generic algorithm. Once again, it is also interesting to compare the same methods for different processes. The plots in figure 26, figure 41, figure 27, figure 42, figure 28, and figure 43 all show that changing the process does not affect the performance of the training algorithm nearly as much as it does the generic algorithm (figure 16, figure 31, figure 17, figure 32, figure 18, and figure 33.)

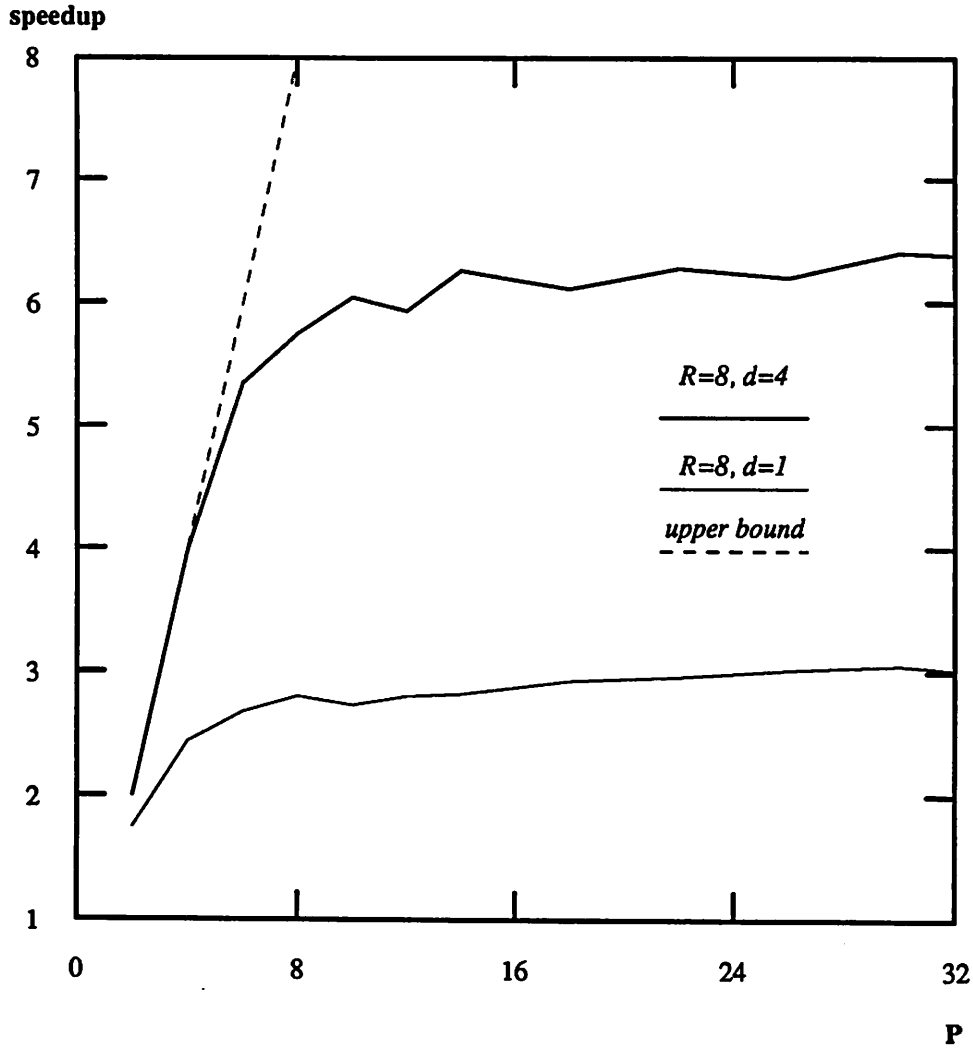

Figure 38 (scorevsP2). Speedup as a Function of Processor/Memory Speed Ratio for the Second Process (Score Based Algorithm)

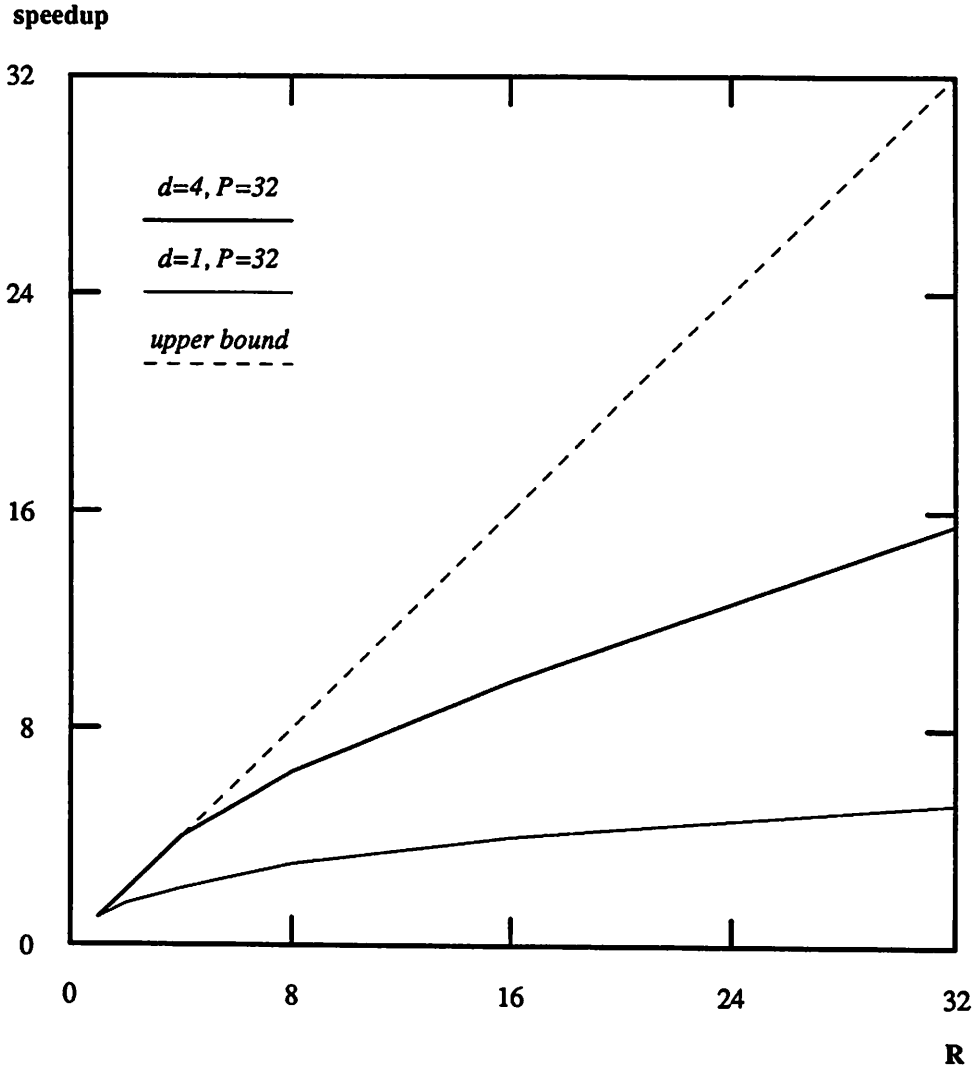

Figure 39 (scorevsR2). Speedup as a Function of Number of Banks for the Second Process (Score Based Algorithm)

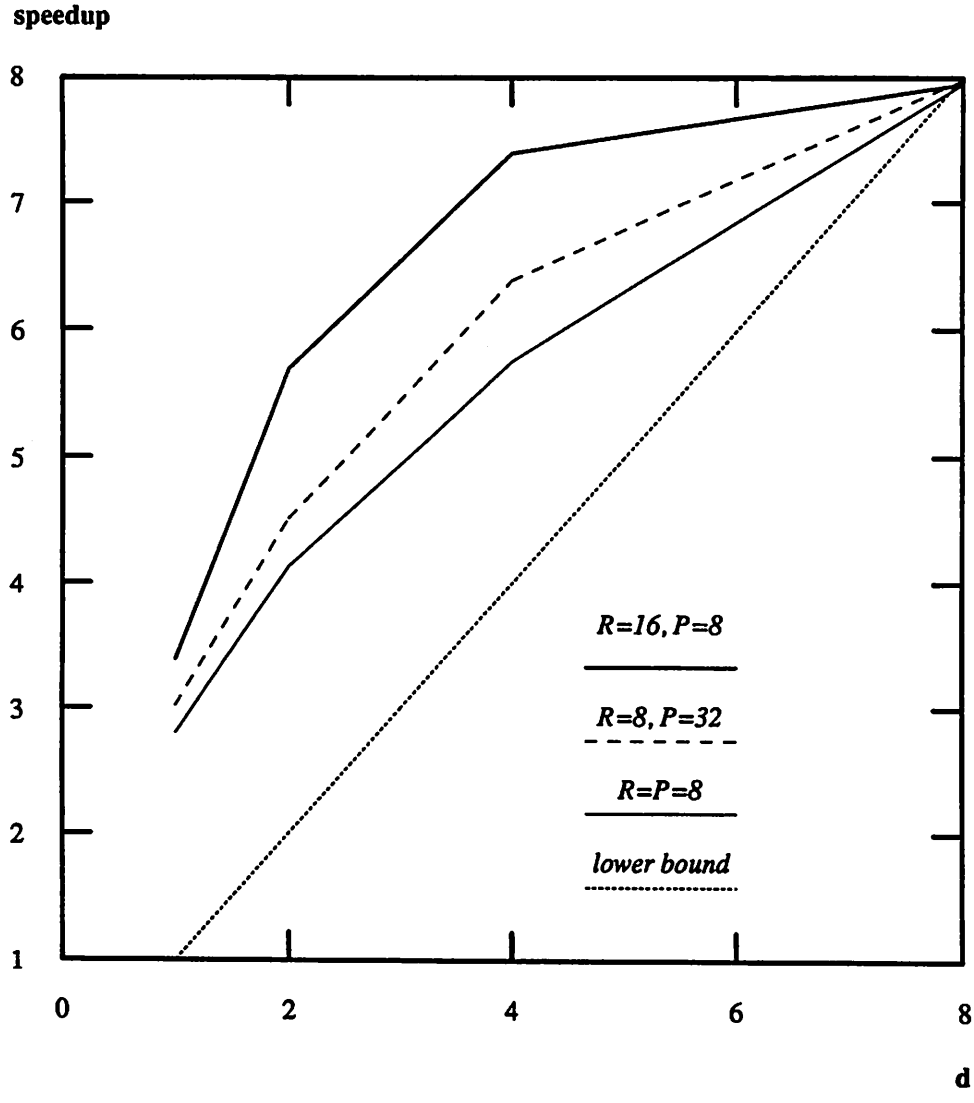

Figure 40 (scorevsd2). Speedup as a Function of Duplication for the Second Process (Score Based Algorithm)

Again, this suggests greater robustness in the training algorithm, and that the training algorithm adapts the allocation to the process.

5.2.5. The Use of Cache

Cache was simulated for the second process with the score based and training methods as well. Speedup is plotted as a function of cache size for different situations for the score based and training algorithms in figure 44. When there is no duplication, we see once again

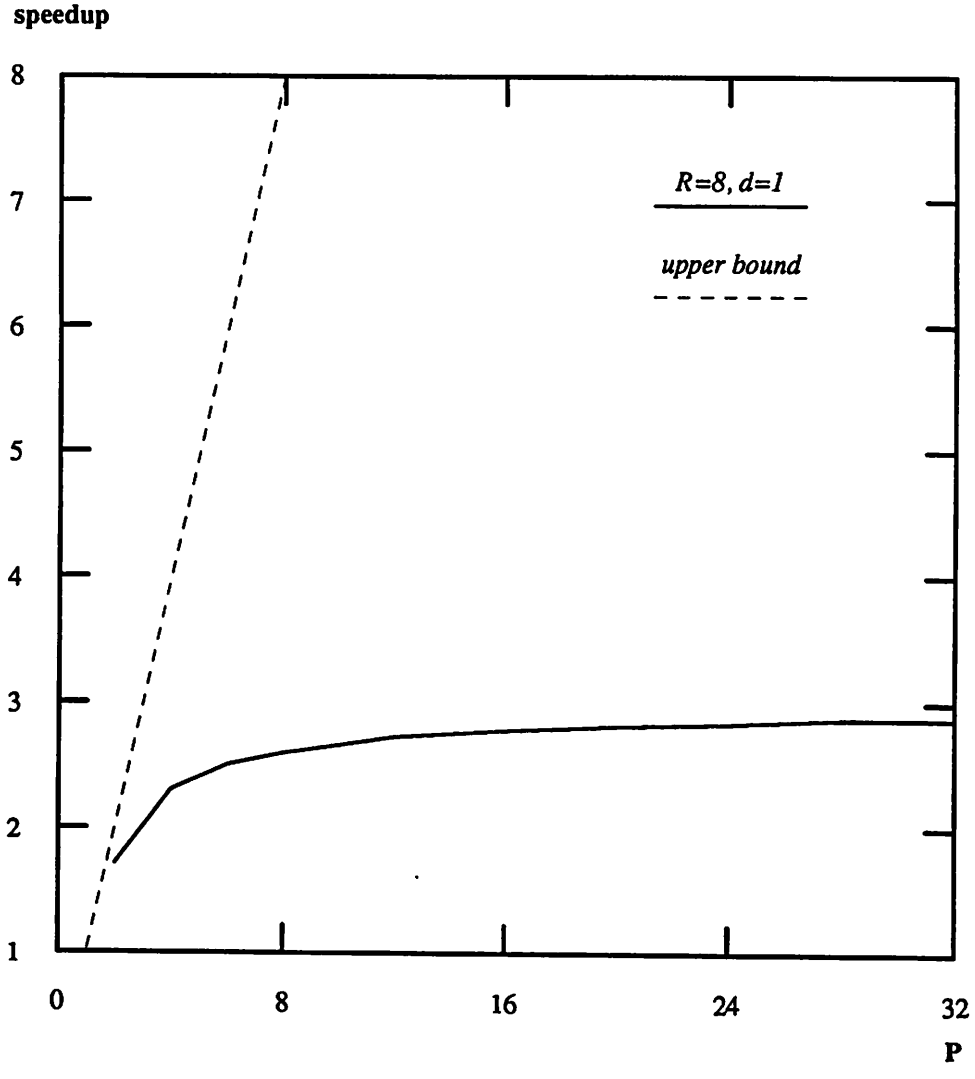

Figure 41 (train1vsP2). Speedup as a Function of Processor/Memory Speed Ratio for the Second Process (Training Algorithm)

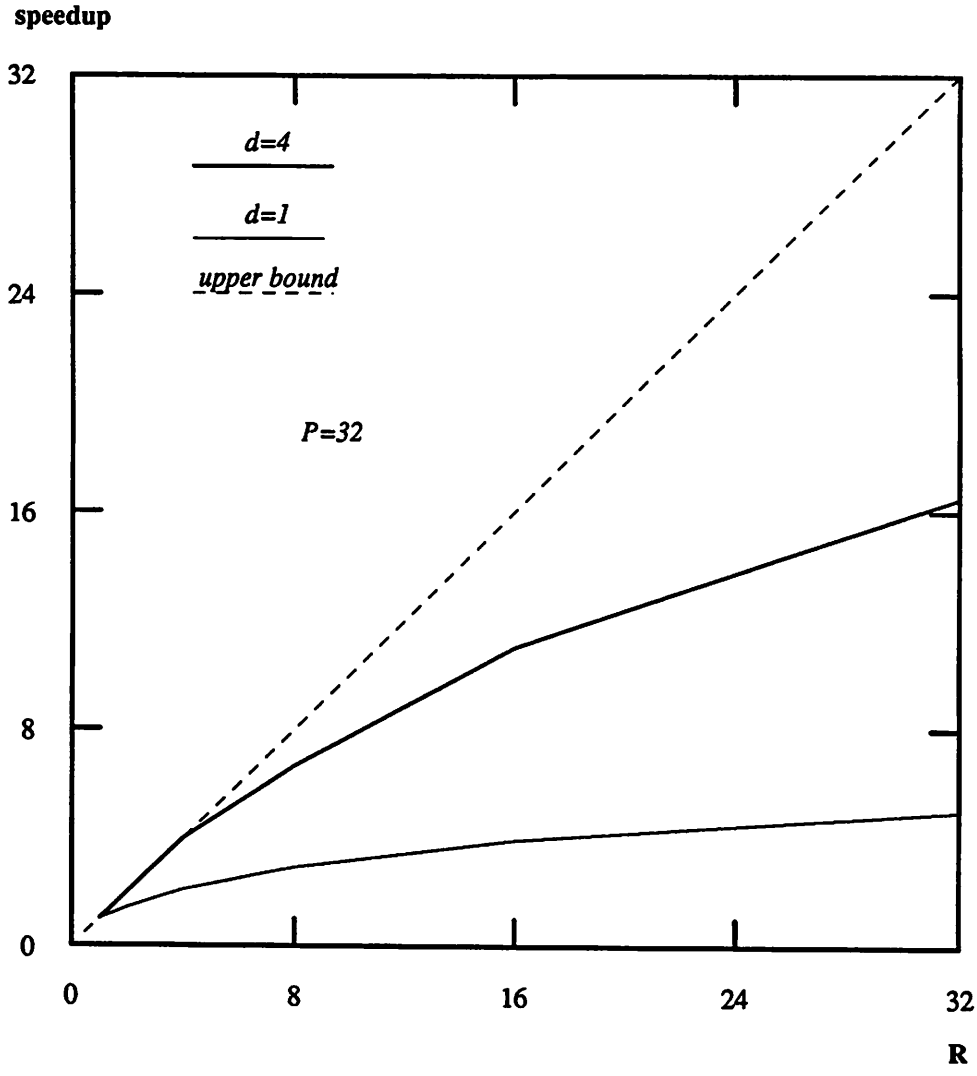

Figure 42 (train1vsR2). Speedup as a Function of Number of Banks for the Second Process (Training Algorithm)

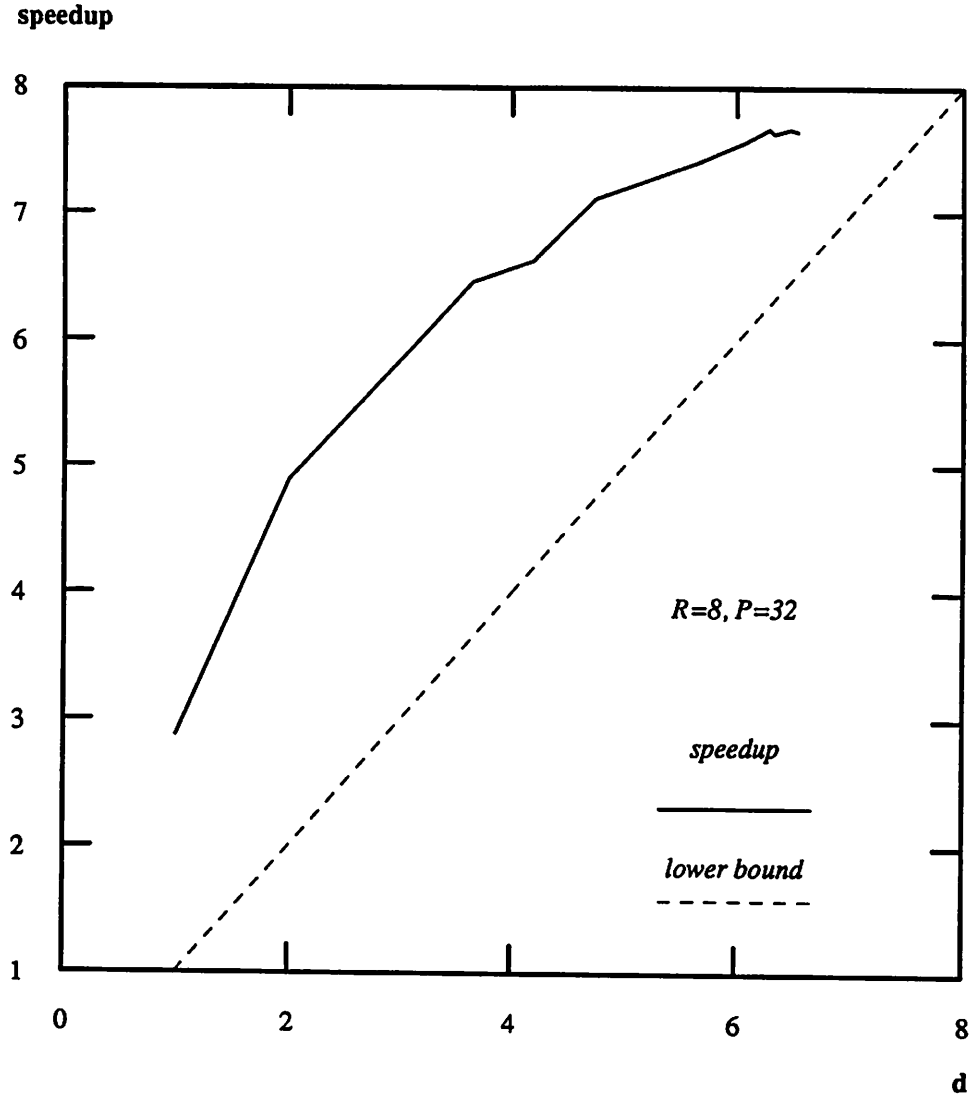

Figure 43 (trainvsd2). Speedup as a Function of Duplication for the Second Process (Training Algorithm)

that cache is not very helpful unless it is used in large amounts. We see a slight improvement over the first process for the case with duplication of 4. The larger slope indicates that additional amounts of cache are more worthwhile in terms of providing greater speedup. The slope is increasing, however, indicating that one must have large amounts of cache to begin with before the additional amount of cache is worthwhile. Some improvement is expected in general for the second process because there is more likely to be temporal locality in the

access. However, the temporal locality is not enough to justify the use of cache. This is exactly the type of process where memory interleaving and proper allocation are called for.

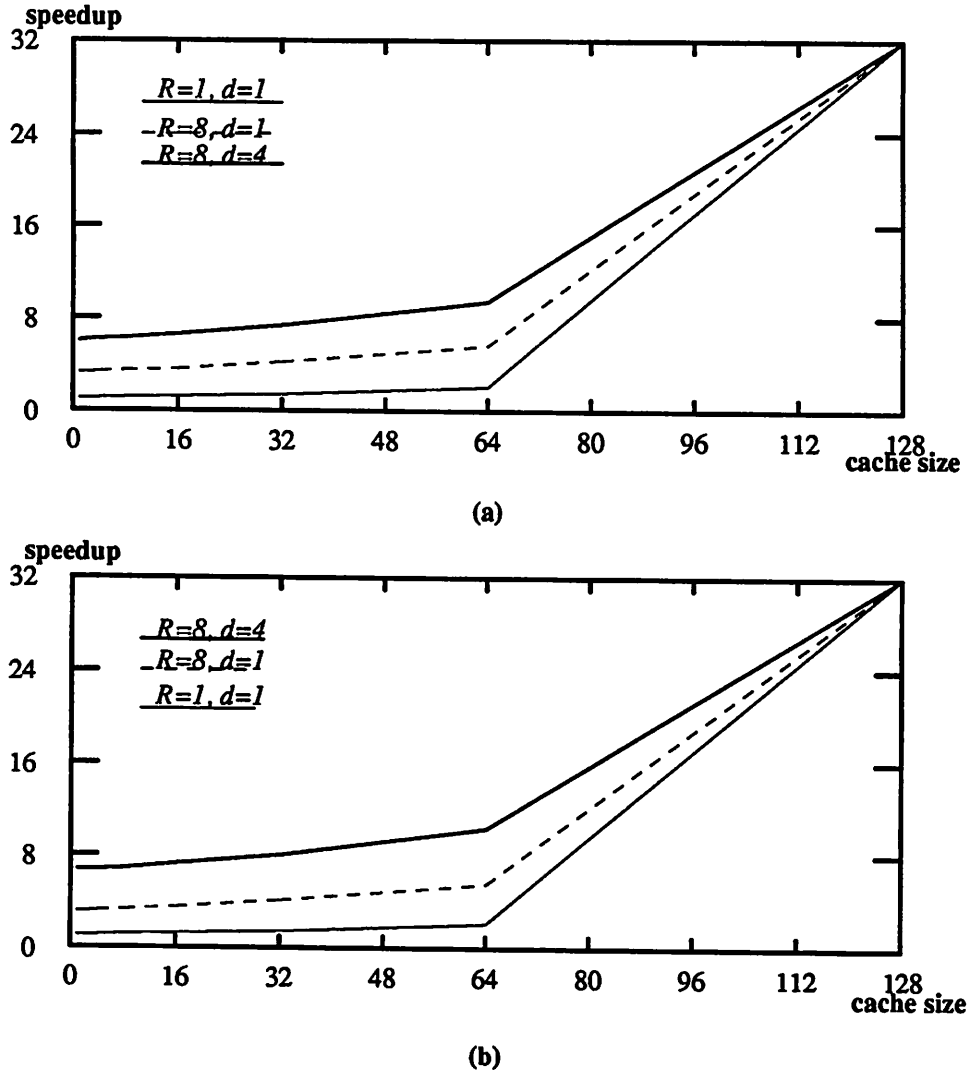

Figure 44 (cache2). Speedup as a Function of Cache Size for the Second Process for (a) Score Based and (b) Training Algorithms

5.3. Summary and Comparison

From the above results, it is evident that the score based and training methods are superior. The Huffman code based method is very elegant and simple, but its rigid scheme for duplication and failure to address temporal conflicts make it undesirable in some applications. Meanwhile, although it can achieve substantial speedup, the sequential access method is limited to a very particular group of processes, and its performance is heavily dependent on the process. Furthermore, the second order statistics needed to perform the allocation are not reliably obtainable, especially with large alphabet sizes and/or short training sequences. Nonetheless, these methods should be viewed as valuable tools which greatly increase our insights into the problem.

Regardless of the algorithm used, the effect of increasing the processor to memory speed ratio, P, was the same. Speedup increased with increasing P, and leveled off after P achieved a certain value.

The last two methods achieve good results and do not demand information which is hard to come by. The score based method directly addresses the temporal conflict issue, and therefore is suitable for a wider range of processes. The training method takes yet a more direct approach by allocating columns from the training sequence as they arrive, making it amenable to real time applications. The derivation of these algorithms did not assume any particular kind of process (although the tests were run using a first order Markov process.)

It must be remembered that both the score based and training algorithms provide an allocation with varying bank sizes. If this allocation is determined before the system is designed, the memory requirement is simply the sum of the bank sizes. However, if the system is designed with a number of banks with fixed size before the allocation is made, there is bound to be some wasted memory space and/or slower performance. The former case is seen more in the design of special purpose DSP systems, whereas the latter case is encountered in more

general purpose applications.

A very important observation from the simulations is that the possible improvement in bandwidth from an interleaved memory system is heavily dependent on the nature of the data access. Naturally, where there is a bias towards some sequences, i.e., there is more inherent information in the data access, there is more room for optimization. The algorithms we presented are an attempt to improve this bandwidth as much as possible. Other techniques can be used in conjunction with this to improve memory bandwidth further.

6. CONCLUSIONS

In this study, we tried to take a closer look at memory interleaving and allocation algorithms to optimize the interleaving process. We started with four algorithms, and concluded that the last two, namely the score based method and the training method, are the most useful and productive of all. The training method performs quite well, but does not offer direct control of duplication. The score based method performance compares to that of the training algorithm, and allows the user to specify exactly how much duplication shall be allowed. The training method, however, does not require any parameter estimation. It is capable of allocating columns to memory as each label from the training sequence arrives, making it amenable to real time applications. Both methods have the advantage that in their development, no information about the process generating the sequence of labels is assumed. Even though the tests in this study used a first order Markov process to generate the labels, this was not necessary for the use of these algorithms. The prime reason for their superiority over the Huffman and sequential access algorithms is the fact that they do not limit themselves to first or second order effects. At the same time, they do not require information which is difficult to obtain, such as high order probabilities.

As a result of this study, we have two algorithms which provide a clear significant improvement in i/o bandwidth of main memory by allocating columns to different banks in a particular fashion. These algorithms are suited for special purpose DSP applications, where the task to be performed is rather well defined, and the memory access pattern has some repetition and inherent information. The testing of these algorithms will be continued in the future real label sequences obtained from a speech processing system. Unfortunately, this sequence is not available at the time of writing.

It should be noted that in the problem formulation, the allocation of *columns* of data, rather than actual individual data, was addressed. Although this was motivated by the speech recognition problem, it is desirable to allocate data in groups simply because of the complexity in keeping track of the locations of data during run time. A relatively small number of columns can be kept track of much easier than the entire set of data. The size of any translation buffer containing the locations of the columns would be at most R*l. Therefore, a smaller l means less complexity in addressing at run time, but also less degrees of freedom in allocation, and less speedup. The grouping of data into columns itself is a problem similar to allocating the different columns into different banks. In order to prevent having to keep track of individual data columns, the columns should not be skewed from the original arrangement of data(e.g. columns of output probabilities corresponding to acoustic labels). If, however, there is no implicit original arrangement, data can be arranged into columns using algorithms similar to the ones presented here. In either case, the choice of l (the number of columns in which data should be organized) is a crucial system parameter.

The performance of the various algorithms was examined as a function of various system parameters, e.g. number of banks, processor to memory speed ratio, and cache size. Although these factors are important, performance is also very dependent on the nature of the process generating the sequence of labels for the system to access. In a process where more

can be predicted about which label arrives given the current label(s), allocating columns one way or another could make a big difference in overall memory bandwidth. In the extreme situation where all labels have the same probability of occurrence regardless of the current label, there is not much to be gained from the allocation algorithms presented here. A possible area for future study is the dependence of the various algorithms' performances on the nature of the process. In such a study, it would be quite helpful to determine certain measures which indicate the "randomness" of the process, and help to give a more accurate and quantitative understanding of this dependence.

REFERENCES

[1] D.G. Messerschmitt, " Breaking the Recursive Bottleneck", *Performance Limits in Communication Theory and Practice"*, ed. J.K. Skwirzynski, Kluwer Academic Publishers, 1988.

[2] H.D.Lin, D.G. Messerschmitt, "Improving the Iteration Bound of Finite State Machines", *IEEE International Symposium on Circuits and Systems Proceedings, Portland, Oregon, May 8 - 11, 1989.*

[3] H.D.Lin, D.G.Messerchmitt, "Finite State Machines have Unlimited Concurrency", *IEEE Transactions on Circuits and Systems*, Dec. 18, 1989.

[4] S.Y. Kung, "Why Systolic Arrays?", *Computer*, Jan. 1982, pp. 37-46.

[5] S.Y. Kung, "VLSI Array Processors", Prentice Hall Englewood Cliffs, 1988.

[6] J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufman Publishers, 1989.

[7] S. Przybylski, M. Horowitz, J. Hennessy. "Performance Tradeoffs in Cache Design",*15th International Symposium on Computer Architecture Proceedings* .

[8] B. Glass. "Caching in on Memory Systems", *Byte* , March 1989, pp. 281 - 285.

[9] A. Averbuch, L.R. Bahl, R. Bakis, P.F. Brown, A. Cole, G. Daggett, S.K. Das, K.Davies, S.V. DeGennaro, P.V. deSouza, E.A. Epstein, D. Fraleigh, F. Jelinek, J. Moorhead, B.L.Lewis, R.L. Mercer, A.J. Nadas, D. Nahamoo, M.A. Picheny, G. Schichman, and P. Spinelli, "Experiments with the Tangora 20,000 Word Speech Recognizer", *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Dallas, Texas*, pp.701 - 704, April, 1987.

[10] L.R. Bahl, F. Jelinek, and R.L. Mercer, "A Maximum Likelihood Approach to Continuous Speech Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, March 1983, pp. 179-90.

[11] F. Jelinek, "The Development of an Experimental Discrete Dictation Recognizer", *Proceeedings of the IEEE*, vol. 73, no. 11, November 1985, pp. 1616-1624.

[12] J. Picone. "Continuous Speech Recognition Using Hidden Markov Models", *IEEE ASSP Magazine*, July 1990,pp. 26-41.

[13] P.Budnick, D.J. Kuck. "The Organization and Use of Parallel Memories", *IEEE Transactions on Computers*, December 1971, pp. 1566 - 1573.

[14] D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proceedings of the I.R.E.* September 1952, pp. 1098 - 1101.

[15] J. Amsterdam, "Data Compression with Huffman Coding", *Byte*, May 1986, pp. 99 - 108.

[16] M.L. Honig, D.G. Messerschmitt, "Adaptive FIlters: Structures, Algorithms, and Applications", Kluwer Academic Publishers, Boston, 1984.

[17] R.M. Gray, "Vector Quantization", *IEEE ASSP Magazine*, April 1984, pp. 4 - 29.

[18] Kai-Fu Lee, "Context Dependent Phonetic Hidden Markov Models for Speaker Independent Continuous Speech Recognition", *IEEE Transactions on ASSP*, vol. 38, no.4, April 1990, pp. 599 - 609.

[19] F. Jelinek, "A Fast Sequential Decoding Algorithm Using a Stack", *IBM Journal of Research and Development*, 1969.

[20] D. Skillicorn, "A Taxonomy for Computer Architectures", *Computer*, Nov 1988.

[21] H.A.G. Wijshoff, "Data Organization in Parallel Computers", Kluwer Academic Publishers, 1989, Boston.

[22] M.E. Mace, "Memory Storage Patterns in Parallel Processing", Kluwer Academic Publishers, 1989, Boston.

[23] D.H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Transactions on Computers*, vol. C-24, no. 12, Dec. 1975, pp. 1145 - 1155.

[24] J. Rabaey, R.W. Broderson, A. Stolze, D. Chen, S. Narayanaswamy, R. Yu, P. Schrupp, H. Murveit, A. Santos,

"A Large Vocabulary Real Time Continuous Speech Recognition

System", ICASSP May 23-26, 1989, Glasgow, Scotland.

APPENDIX

Word Description and Source Code for:

Label Sequence Generation
Memory Access Simulation
Generic Algorithm
Huffman Code Based Algorithm
Sequential Access Algorithm
Score Based Algorithm
Training Algorithm

Label Sequence Generation

matrixgen.c

markovgen.c

error.c

These programs are used to generate label sequences. "Matrixgen.c" generates a Markov transition matrix. "Markovgen.c" generates the actual label sequence, and "error.c" calculates the error between calculated and actual transition probabilities.

"Matrixgen.c" generates a Markov transition matrix for the two processes described in section 5. It first asks the user to determine the process to be used. Then it generates the transition matrix one row at a time. If the first process is desired, the transition matrix elements are generated by a random number generator. Otherwise, the maximum position in each row is chosen to be a 1, and the other positions decay exponentially with a factor also put in by the user. The larger this factor is, the closer the process approaches the deterministic case. In both cases, the row sum is accumulated, and once the entire row has been generated, the row elements are normalized to the row sum to make the matrix meet the necessary criteria for a Markov process. This procedure is repeated until all the rows of the matrix have been generated. The matrix is then written to the file "markov_matrix" one row at a time.

"Markovgen.c" reads the transition probabilities from the file "markov_matrix" and uses a random number generator to create two label sequences of any length for training and testing. It also estimates first and second order probabilities at the end of sequence generation. First, the program reads the elements of the transition matrix

"Q". It then initializes "norm" and "count", two arrays of count variables, to 0. Next, the combined length of the sequences to be generated is requested from the user. In this program, the output equals the state. However, in a Hidden Markov Model, the states are not observed. This decision was made because of the need to have control over the nature of the process. The program arbitrarily sets the first state and the first output to 1. The next state (and output) is determined by generating a number from a random process uniformly distributed over [0,1). This interval is partitioned according to the transition probabilities in the ith row, where i is the current state. The interval in which the random number falls determines the next state. This determination is made in a simple for loop. At this point, the *current state* is written to the appropriate file ("markov_output.out" for the first half of the sequence, or the training sequence, and "output1.out" for the test sequence), and the appropriate count variables are incremented. This procedure is repeated until the label sequence is as long as the user specified it to be. Finally, the first and second order probabilities are estimated from the appropriate count variables and are put out to separate files ("stats1.out" and "stats2.out").

"Error.c" is a simple program to calculate the average and maximum error between estimated and actual second order probabilities. It simply accumulates the absolute values of the differences between the estimated values from "stats2.out" and the actual transition probabilities from "markov_matrix", and divides this sum by the number of terms, that is, the alphabet size squared. During the evaluation of these terms, it also keeps track of the maximum error.

Memory Access Simulation

memsim.c

This program simulates the access of an interleaved memory system. It reads item locations from a file named "allocation.data". It then reads a long label sequence from a file named "markov_output.out". As each label occurs, it searches for the first available bank which has the corresponding item. Each bank has a "ready time" which is constantly updated to account for the bank's latency. Time is measured in terms of processor cycles, and is simply called "cycles". There are also count variables to keep track of number of times a bank was accessed, number of times a bank was waited for, and how much duplication there is in the allocation scheme. In the beginning of the program, everything is initialized. The cache is filled with arbitrary items. The program first asks the user to determine whether cache should be used. Then, allocation of data is determined. The locations array is determined from the file "allocation.data". The locations array in this program is called "Bank". The location determination process moves to the item corresponding to the next label in the alphabet either when the number of locations read equals the number of banks, or a -1 is encountered in the file "allocation.data". Once the locations array is filled, the program requests the value of "data_length", or the length of the label sequence desired to be used in the simulation. Now, the program is ready to start simulation.

There are two possible sources for labels: the main sequence of labels in the file "markov_output.out", and an internal "stack". A variable named "wait" indicates whether data is in the stack or not, and therefore where the next label should be read

from. Once the label is read, another variable named "stack_flag" indicates the origin of the current label. The variable "input_time" indicates the when data was entered into the stack. If data is read from the stack, the variable "wait" is decremented, indicating an empty stack. Before the search for the proper item begins, a check is made to see if the cache should be updated. This procedure will be explained shortly. Once the current label is specified, the next step is to search for the corresponding item.

The search for the appropriate item begins in the cache, if cache is being used. If cache is not being used or the search in the cache was fruitless, the search continues in main memory. A "read" from cache is simulated by setting the variable "in_cache" equal to 'y', indicating that the item is in fact in the cache. Also, the last time the particular cache location was used is updated for later use in determining the LRU. The variable "datacount" is also incremented, indicating that one more label has been processed and the simulation is ready for the next label. The search in main memory is simulated with the help of the locations array, known in the program as "Bank". Each location is tried. If the current time is before the time the location is ready (i.e. the location is busy), the next location is tried. If all locations with the desired item are busy, the label is placed in the "stack", and "wait" is incremented to indicate that there is data in the stack. This in effect, halts the input of labels from the main label sequence and introduces a wait state. The wait for the current item can be found by subtracting the input time, i.e. the time the label entered the system from the main sequence, from the current time, and adding 1 to count the current cycle. This wait is always calculated to keep track of the maximum wait occurring in the

simulation. The total wait is also incremented each time this occurs. The total wait is kept track of for later use. If one of the locations being checked is in fact available, a "read" from the main memory is simulated. This is achieved by simply incrementing "datacount", incrementing "count", which keeps track of how many times the location was accessed, and updating the next available time for the location.

Before moving to the next processor cycle and label search, it must be determined whether or not to place the current item in cache. If the current item is not in cache, it is written to the least recently used, or LRU position in cache. The LRU is determined by a simple for loop. The LRU is labeled, but it is not written to immediately, due to the latency of the main memory. Three arrays are used to assist in this delayed write: "days_are_numbered", "time_of_request", and "temp_contents". "Days_are_numbered" indicates if a location in cache is targeted for a write. "Time_of_request" indicates the time this determination was made. "Temp_contents" holds the data to be written to the cache location once the latency period is over. In the beginning of the process, a check is made to see if a cache location is to be overwritten. This check is simply made by comparing the current time to time of request +ratio, and also checking for the flag variable "days_are_numbered".

The simulation terminates once datacount equals data length, the simulation length specified by the user. At the end of the simulation, the overall number of cycles is adjusted to account for the memory banks' latencies. If main memory was not used at all (everything happened to be in cache), the cycles are not changed. Otherwise, the cycles are incremented by ratio -1 to account for the memory latency.

Finally, the program puts out the final results. The outputs are the overall simulation length in number of labels, total number of processor cycles, total number of waiting cycles, maximum wait, speedup, duplication, and the number of times each individual bank was either accessed or tried but busy.

Generic Algorithm

randomall.c

This program allocates item according to the "generic" algorithm described in section 4.2. In this algorithm, items are successively inserted into banks, continuing at the next bank when the current bank is filled. When there are no more items, allocation starts over at the first item. This process continues until all banks are full. The bank size must divide into the alphabet size, and the bank size times the number of banks must be an integer multiple of the alphabet size.

The program starts by asking the user for the amount of duplication required. Since the number of banks and alphabet size are already known, the individual bank size can then be easily calculated. After this calculation, the program begins allocation. Allocation is made using a locations array, and it is easily seen that the locations for each item i are integer(i/ bank size) + (j*number of banks/duplication), where j is an integer varying from 0 to duplication -1. If duplication is not equal to the number of banks, a -1 must be inserted at the end of the locations of each item to maintain the format required by the simulation program "memsim.c". The program in fact takes all these actions and writes the results to the file "allocation.data".

delim $$

Huffman Code Based Algorithm

huffman.c

This program allocates data items according to the Huffman code based algorithm described in section 4.3. For this purpose, it needs to construct a binary tree as shown in figure (). The variable structure "node" is defined in the beginning of the program. The elements of this structure are a number indicating the node probability called score, a character indicating whether the node is a source node , which will be described later, an integer called label, another integer called source_no, a left child, a right child, and a previous node. In a Huffman tree, there are two kinds of nodes: leaves and internal nodes. Leaves have parent nodes but no children. All other nodes are called internal nodes. A node is a source node if all the leaves originating from that node are in one bank. Also, it is helpful to know that in a Huffman tree, if there are l leaves, there are l internal nodes. In the beginning of the program two arrays of type node are declared. One of these arrays represents the leaves, and has size l. The other represents the internal nodes, and has size l -1.

During initialization, all nodes are isolated. That is, they are their own children as well as their own parent. They are also assigned a very high score of 10. These will be changed during the construction of the Huffman tree.

The Huffman tree is constructed in l -1 iterations. There is an "active list" of nodes being considered in each iteration for their individual scores. The size of this list decreases with more iterations, but this is taken care of by filling in the unused

positions in the active list with "dummy" nodes. The dummy nodes have a high score and are guaranteed not to be chosen. First, an array of scores of nodes in the active list is created. Then, this array is sorted. The first two elements of the sorted array have the lowest score, and are the children of the new internal node. After determining whether the two children are internal nodes or leaves, the proper connection assignments are made regarding the appropriate node's left and right child, and the parent node. The new internal node's score is calculated by adding the scores of the two children. The first two nodes in the active list are replaced by the new internal node and a dummy node, and the next iteration begins. As mentioned before, there are l -1 iterations.

At this point, it is necessary to determine the "source nodes". A node is called a source node if all the leaves originating from that node reside in the same bank. This determination is made by looking at the depth of the node in the Huffman tree. For this purpose, the leaves array and the internal nodes array are combined into one large array. The depth of each node is then measured by counting how many nodes must be passed before reaching the root node. The depth of the root node is set to 0. At this point, all nodes which have a depth of $log sub 2 R$, where R is the number of banks, are considered to be a source node. This change is made in the original separate arrays of leaves and internal nodes. If the depth of a leaf is less than $log sub 2 R$, that leaf will be duplicated in the allocation. The amount of duplication is R/$2 sup depth$, where depth is the depth of the leaf.

Now, the allocation is conducted directly from the Huffman tree. A pointer is put on each leaf. It traverses the tree upward until it encounters a source node. The

number of that source node is the bank location of the leaf. This is repeated for each leaf. If a leaf is duplicated, several locations are assigned to it, according to its duplication. Finally, the program outputs the locations array to the file "allocation.data" in the format required by the file "memsim.c".

Sequential Access Algorithm

seq_organize.c

seq_allocate.c

interleave.c

The allocation for the sequential access method is done by the files "seq_organize.c" and "seq_allocate.c". "Allocate.c" is the main routine, and calls upon "seq_allocate.c" to do the actual allocation. The allocation is defined by two approaches. The first one is to define a two dimensional array containing the locations of each item. One dimension of this array is the alphabet size and the other is the number of banks. The other approach is to define a two dimensional array showing the contents of each bank. The dimensions of this array are the number of banks and the maximum bank size. The first approach is needed for use by the main simulation program "memsim.c". However, we have written a separate program to simulate sequential access called "interleave.c", and this program requires the second format.

In "seq_organize.c", these two arrays are called banks and in_banks. Some of the elements of these arrays are empty during or after running of the program. The convention is that "empty" elements are assigned a value of -1. The first part of "seq_organize.c" initializes all array elements to -1 (empty), and reads the (estimated) transition probabilities from the file "stats2.out". Then, the file "seq_allocate.c" is called upon to actually allocate the items using the sequential access algorithm. Next, "seq_organize.c" checks each bank for any duplicated items within that bank (

This is a troubleshooting step). Finally, it puts out the locations array to the file "allocation.data", and the contents array to the file "contents.banks".

The file "seq_allocate.c" actually implements the sequential access algorithm. In this file, the contents array is called "contents", and the locations array is called "locations". The file starts working with the locations array, and initializes all its elements to -1 (empty). The other key variables are "testprobs", which is an array of probabilities evaluated for each item from step iii of the algorithm (section 4.3), and "flag", a two dimensional array of flags indicating whether a given item exists in a given bank. Obviously, the flag array can be evaluated from the locations array and vice versa. Another array called "number_of_entries" keeps track of how many items have been allocated to each bank. These variables are also initialized.

The first entry in the first bank is set to the item with highest probability of occurrence. To evaluate the first entry in the next bank, "testprobs" is evaluated, and the item with the highest "testprobs" is allocated to the next bank. Before this is done, however, a check is made to see whether the item already existed in the bank. If so, the item with the next highest "testprob" is put in (provided it is not already in the bank). This check is made with a simple while loop. When the contents are "written", the corresponding "number_of_entries" and "flag" variables are also updated. When the number of banks is reached, the next "row" is processed starting at the first bank. This boundary condition is handled in a similar way to the normal case, keeping in mind that the previous bank is the last bank, and the next bank is the first bank. To avoid boundary complications, the last iteration is also handled separately.

At this point, the locations array is derived from the contents array. If all items were guaranteed to have been allocated, allocation would be complete. However, there may be some unallocated items, which the program checks for next. After determining the unallocated items, the program rounds up all the items in the first bank, and sorts them in order of their probability of occurrence. Then, it checks each one starting with the least probable item to see if it is located elsewhere. If so, it is replaced with the current unallocated item. Otherwise, to search continues to the next to least probable item, and so on. Only one unallocated item is allowed to be written in a bank at a time. The process is repeated for the next bank, and so on, until all the unallocated items have been allocated.

Finally, the locations array is updated using the contents array. It is easily seen that the locations array can be obtained from the contents array and vice versa.

The program "interleave.c" is a special program written to simulate memory accessing for the sequential access algorithm. The memory locations are no longer determined by a locations array, but instead from a contents array. The dimensions of this array are number of banks and bank size. This array is read from the file "contents.banks". Like the program "memsim.c", this program also prompts the user for the length of the label sequence to be simulated. Then, the other quantities in the program are also initialized. "Flag_empty" is a flag variable which indicates whether the bank encountered has the desired item or not. "Total_wait" is the total number of cycles the processor had to wait for items to be accessed. "Wait" is an array indicating the total number of wait states for each bank. "Ready_time" is an array of available times for each bank. "Bank" simply is the current bank being tried, "cycles" is

the time measured in processor cycles, and "count" indicates how many items have been retrieved so far.

The simulation begins at the first bank. Each cycle, the bank is chosen sequentially. If no item is currently being waited for, the next label is read from the file "markov_output.out". "Flag_empty" is automatically set to 1. If the current bank is available, and if the current bank contains the desired item, "flag_empty" is reset to 0, the ready time of the bank is updated, and "count" is incremented. If the desired item was not obtained in this cycle, the wait variables are incremented, and the simulation moves on to the next iteration. This process continues all the items corresponding to all the labels in the main sequence have been obtained. At the end of the simulation is calculated. Finally, the program outputs the total number of cycles, the individual and total number of wait states, the speedup, and the overall duplication.

Score Based Algorithm

analyze.c

score_based.c

The file "analyze.c" analyzes the label sequence in the file "markov_output.out" and calculates the scores necessary for the score based method. The program initializes all the count variables and time variables. Then it reads the labels one by one. As it does this, it compares the last time of occurrence of each of the labels, and if it is within P cycles of the current time, the appropriate count variables are incremented. Also, the last time of occurrence of the label that was just read is also updated. Then, the count matrix is made symmetric by first adding the symmetric elements together and setting the symmetric elements equal. Therefore, the score calculated is a simple count. Finally, the score matrix is output to the file "score.out", ready for use by the program "score_based.c".

The actual allocation is done by the program "score_based.c". As described in section 4.5, the score based method only allocates for duplications of 1. Higher duplication can be achieved by using fewer banks, and then duplicating the scheme. The program first requests the allowed duplication from the user. It then reduces the number of banks accordingly. Initialization takes place next. All the count variables are set to 0, all the contents are set to -1 (empty), and the locations array is also set to -1 (empty). "Max_count", which keeps track of the maximum bank size is set to 0. "Item" refers to the current item under consideration, and it is also set to 0. Then, the program reads in the scores for all possible pairs in the alphabet.

At this point, the allocation starts. Allocation is done item by item. As each item is considered, a test is run on all the banks. For each bank, the maximum score resulting from adding the new item is evaluated. These maximum scores are stored in the array "temp", and the bank indices are stored in the array "index". These arrays are sorted, and the bank with lowest maximum score is chosen as the location for the current item. The next element in the contents matrix for the chosen bank is assigned as the current item, and the proper count variable is incremented. At this point, the maximum bank size is kept track of in the variable "max_count", and allocation moves the next item.

At the end of the allocation, the maximum bank size is put out, and the locations array is derived from the contents array. Finally, the locations array is put out to the file "allocation.data" in a slightly modified way. The modification takes place to accomodate the duplication required at the beginning.

Training Algorithm

training.c

The training algorithm is implemented by the program "training.c". This algorithm is described thoroughly in section 4.6. During the allocation, the program keeps track of various quantities, such as the number of locations assigned to an item so far, the first available read location for an item, the first available write location for an item, and the next available time for all the banks. There are also some flag variables indicating whether an item has been allocated (the array "allocated"), and whether all banks are full ("all_banks_full"). In the beginning, all of these parameters are initialized. In this program, both locations and contents arrays exist, and they are also initialized to all -1's in the beginning. Next, the program asks the user to specify a "maximum wait" beyond which waiting is not desired. A lower maximum wait results in more duplication.

At this point, the simulation begins. The first step is to sort the banks in terms of their ready times. This is to facilitate determining of earliest read and write locations later. Next, a label is read from the training sequence. The next available location that the corresponding item can read from is determined in a nested while loop. The flag "already_there" is assumed to be equal to 'n', until a search through the banks in their sorted order shows otherwise. If this happens, the flag is set to 'y', and the next available read location and its next available time are kept track of. A similar procedure is used to find the next available write location for the desired item as well as the corresponding time. A search is conducted through all the banks in order

of their availability using a while loop and a for loop. The last iteration had to be separated from the main while loop to avoid exceeding array argument bounds.

This information allows the program to proceed with the necessary decision making. There are three cases: the item has not yet been allocated, the item has been allocated but must be waited for too long and the next available write location happens earlier than the next available read location, or the item has been allocated and does not have to be waited for too long. In the first case, the item must obviously be allocated to the soonest available bank. This is indeed what the program does. It assigns the item to the contents array, and the new location to the locations array, and it increments the necessary count variables. It also increments the ready time of the bank according to the current time and the previous ready time of the bank. If it was ready before the current time, the new ready time is simply P cycles from now. Otherwise, the current time must be updated to the ready time of the bank (resulting in a wait period), and the new ready time is the old ready time + P cycles. This ready time is also the next available read time of the item since it is located nowhere else. Finally, the "allocated" flag is asserted to avoid repeating this case in future iterations.

The second case is exactly like the first case, except that the "allocated" flag need not be changed, and a check must be made to see if the next available read location is available later than the next available write location. If so, a write is initiated. Otherwise, the current time is simply updated to the ready time of the next available read location, and the wait is recorded.

The third case requires no write operation, and the necessary steps are taken to simulate a read operation. In the final part of each iteration, the maximum wait ("longest_wait") is updated, and a check is made to see if all banks are full yet. This entire process is continued until all banks are full, or the end of the training sequence is reached. Finally, the locations array is put out to the file "allocation.data" in the format required by the program "memsim.c".