C. K. Wong
D. T. Tang

Dynamic Memories with Faster Random and Sequential Access

Abstract: This paper extends the dynamic memory proposed by Aho and Ullman in two directions. Instead of a shuffle permutation, block shuffles are introduced. By choosing suitable block sizes, faster random and sequential access may result. Another direction of extension comes from the addition of a reverse cyclic permutation, which results in even faster random and sequential access. Analyses for both the worst and the average cases are given. Generalizations to arbitrary radices are also discussed.

Introduction

In [1-3], a special kind of memory, called *dynamic memory*, is studied. It is an array of n cells, each of which can hold one data word. The contents of only one of the cells, called the window, can be read or written externally. Throughout this paper, the window is assumed to be cell 0. (See Fig. 1.) However, the contents of the cells in the memory can be rearranged internally by applying a sequence of operations called *memory transformations*. Each transformation is a permutation of the contents of the n cells. For example, if the permutation $\pi = (i_0, i_1, \cdots, i_{n-1})$ is applied to an n-cell dynamic memory, then in one step the contents of cell j are transferred to cell i_j for all j, $0 \le j < n$.

In [3], the following two transformations are proposed ($n = 2^k$):

$$\pi_t(i) = \begin{cases} 2i, & i = 0, 1, \cdots, \frac{n}{2} - 1, \\ 2i - n + 1, & i = \frac{n}{2}, \frac{n}{2} + 1, \cdots, n - 1, \end{cases} \tag{1}$$

and

$$\pi_{et}(i) = \pi_t(\pi_e(i)),$$

where

$$\pi_e(i) = \begin{cases} i + 1, & i \text{ even}, \\ i - 1, & i \text{ odd}. \end{cases} \tag{2}$$

By means of these two transformations, the contents of any cell can be brought to the window in no more than $k = \log_2 n$ steps. Suppose datum j denotes the initial contents of cell j. To access a block of data, say datum j, datum $j + 1, \cdots$, datum $j + b - 1$, we must first locate the cell that contains datum j. Then we must locate the cell that contains datum $j + 1$, and so on. Therefore, in the worst case, $b \log_2 n$ steps are needed. However, Stone [5] has recently shown that, for certain sizes of this kind of memory, an address recoding exists that results in fast sequential access as well as random access. For other work related to these transformations and dynamic memories, see [6-11].

In [1], another pair of transformations is proposed for $n = 2^k - 1$, by which any datum j can be accessed in no more than $2 \log_2(n + 1) - 2$ steps and any block of $b \ge 2$ data items can be accessed in no more than $3 \log_2(n+1) + b - 4$ steps. These transformations are

$$\pi_m(i) = (2i)_{\bmod n}, \tag{3}$$

$$\pi_s(i) = (i - 1)_{\bmod n}, \tag{4}$$

$i = 0, 1, \cdots, n - 1$, where $x_{\bmod y} = x - y\lfloor x/y \rfloor$. These transformations are further generalized to arbitrary radices [1].

In this paper, we first replace (3) with $\pi_m^{(d)}(i) = (i \cdot 2^{k-d})_{\bmod n}$, $i = 0, 1, \cdots, n - 1$, where d is a parameter that divides k. The resulting memory has the same random and sequential access characteristics as before. In particular, when $d = 2$ and k is even, random access takes no more than $2 \log_2(n + 1) - 2$ steps for the worst case and takes $1.25 \log_2(n + 1) - 1.333$ steps for the average case, which is about a 17 percent improvement. For sequential access, after the first two items have been accessed, each successive item can be accessed in one step as in the original memory.

Later we introduce one more transformation to the memory structure considered above:

$$\pi_a(i) = (i + 1)_{\bmod n}. \tag{5}$$

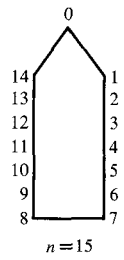

Figure 1 Dynamic memory having 15 cells.

Clearly, this is π_s in the reverse direction. When $d = 2$, random access now takes $1.5 \log_2(n + 1) - 1$ steps for the worst case and $\log_2(n + 1) - 1.333$ steps for the average case (25 and 33 percent improvements, respectively, over the original memory). Finally, generalizations of these two types of memories to arbitrary radices are also discussed.

As pointed out in [3], implementation of transformations in memories such as metal oxide semiconductor (MOS) shift register memories amounts to the embedding of corresponding interconnecting patterns. However, for more rigidly structured memories such as magnetic bubble memories [12, 13] such an embedding is much harder to realize. In [12–15], various ways to introduce some simple forms of memory transformations in magnetic bubble memories are discussed and analyzed. But the access time is far from the logarithmic growth with memory size achieved in [1, 3, 5] and here. In fact, it is basically linear. In view of the rapid growth of this new type of memory, it may be of interest to study new memory transformations, which may reduce the access time, say, from a linear increase to a square root or even a logarithmic increase with memory size.

Since most of the derivation of the formulae is very complicated and is given in research report [4], it is omitted here.

Preliminary considerations

To facilitate later discussion, we restate here some of the results in [1]:

1. A *memory address map* is defined as a function f from data to cells such that $f(j)$ is the cell currently containing datum j. If only transformations π_m and π_s are used, the memory map always has the form

$$f(j) = (j \cdot 2^p + q)_{\bmod n} \tag{6}$$

for integers p and q, where $0 \leq p < k$ and $0 \leq q < n$. (Recall that $n = 2^k - 1$.) Also

$$\pi_s \text{ maps } \begin{cases} p \text{ into } p, \\ q \text{ into } (q-1)_{\bmod n}, \end{cases} \tag{7}$$

$$\pi_m \text{ maps } \begin{cases} p \text{ into } (p+1)_{\bmod k}, \\ q \text{ into } (2q)_{\bmod n}. \end{cases} \tag{8}$$

Initially, the memory map is $f(j) = j$, characterized by $p = q = 0$.

2. An *accessing sequence* for datum j is defined as a sequence of π_m's and π_s's that will move datum j into the window.
3. If we represent $f(j) = (j \cdot 2^p + q)_{\bmod n}$ as a k-bit binary number, the effect of applying π_s to the memory is to subtract 1 from $f(j)$, whereas the effect of applying π_m is to rotate $f(j)$ to the left cyclically by one bit. And application of an accessing sequence to the memory reduces $f(j)$ to 0.

With this understanding, Algorithm 1 in [1] can be informally described as follows:

Algorithm 1 Given p, q (the memory map) and j (the datum to be fetched), this algorithm generates an accessing sequence. As a side effect, p and q are updated so they continue to represent the memory map.

1. Let the binary representation of $f(j)$ be $b_k \cdots b_2\ b_1$.
2. If all $b_i = 0$, stop.
3. If $b_1 = 0$, go to 4). Otherwise generate π_s.
   $b_1 \leftarrow 0$.
   $q \leftarrow (q-1)_{\bmod n}$
   Go to 2).
4. Generate π_m.
   $c \leftarrow b_k$.
   For $i = k, k-1, \cdots, 2$, $b_i \leftarrow b_{i-1}$; $b_1 \leftarrow c$.
   $p \leftarrow (p+1)_{\bmod k}$, $q \leftarrow (2q)_{\bmod n}$.
   Go to 3).

From this algorithm, we have the following results. (For the proof refer to [1].)

• *Theorem 1*
Algorithm 1 brings datum j to the window in no more than $2 \log_2(n + 1) - 2$ steps, where a step means one application of either π_m or π_s. (In other words, the total number of π_m's and π_s's generated by Algorithm 1 is never more than $2 \log_2(n + 1) - 2$.)

• *Theorem 2 (sequential accessing property)*
Suppose we have just used Algorithm 1 to bring datum j and datum $j + 1$ to the window. Then the accessing sequences for each of datum $j + 2$, datum $j + 3, \cdots$ are of length 1, namely, π_s.

Remark In fact, for datum $j + 1$, Algorithm 1 generates only π_m's followed by a single π_s, and thus accessing datum $j + 1$ requires less time than usual. A detailed calculation shows that any block of $b \geq 2$ data can be accessed in no more than $3 \log_2(n + 1) + b - 4$ steps [4].

A class of memory transformations

In this section, we introduce a class of memory transformations, which are generalizations of (3). First, note that if we use the multiplier 2^{k-1} instead of 2 in (3),

$$\pi_m^{(1)}(i) = (i \cdot 2^{k-1})_{\bmod n} \tag{3'}$$

(the superscript (1) will become clear later), all the results in the previous section remain valid since the only difference is that $\pi_m^{(1)}$ now corresponds to a right cyclic rotation of the binary representation of $f(j)$ instead of a left one. However, such a change makes later analysis easier. Second, instead of $k-1$, we use $(\alpha-1)d$ as the exponent in (3'), where $k = \alpha d$ for positive integers α, d. (Recall that $n = 2^k - 1$.) Here d serves as a parameter. Different values of d result in different performance. The choice of d is discussed later. We therefore have a class of transformations,

$$\pi_m^{(d)}(i) = (i \cdot 2^{(\alpha-1)d})_{\bmod n}, \tag{9}$$

parameterized by the integer d. For $d = 1$, (9) becomes (3').

Note that $\pi_m^{(d)}$ now corresponds to a right cyclic rotation of the binary representation of $f(j)$ through d bits instead of 1.

If we use (4) and (9) only, then the memory map takes the form

$$f(j) = (j \cdot 2^{dp} + q)_{\bmod n} \tag{6'}$$

for integers p and q, where $0 \le p < \alpha$ and $0 \le q < n$. Also,

$$\pi_m^{(d)} \text{ maps } \begin{cases} p \text{ into } (p-1)_{\bmod \alpha}, \\ q \text{ into } (q \cdot 2^{(\alpha-1)d})_{\bmod n}. \end{cases} \tag{8'}$$

Corresponding to Algorithm 1, we have the following algorithm.

Algorithm 1' Given p, q (the memory map) and j (the datum to be fetched), this algorithm generates an accessing sequence of π_s's and $\pi_m^{(d)}$'s and updates p, q.

1. Let the binary representation of $f(j)$ be $b_k \cdots b_2\, b_1$. Divide it into α blocks of d bits each. Number the blocks $1, 2, \cdots, \alpha$ from right to left. Let the numbers represented by the blocks be $a_1, a_2, \cdots, a_\alpha$, respectively.
2. If all $a_i = 0$, stop.
3. If $a_1 = 0$, go to 4). Otherwise, generate a_1 π_s's.
   $a_1 \leftarrow 0$.
   $q \leftarrow (q - a_1)_{\bmod n}$.
   Go to 2).
4. Generate $\pi_m^{(d)}$.
   $a_i \leftarrow a_{i+1}$, for $i = 1, 2, \cdots, \alpha - 1$; $a_\alpha \leftarrow 0$.
   $p \leftarrow (p-1)_{\bmod \alpha}$, $q \leftarrow (q \cdot 2^{(\alpha-1)d})_{\bmod n}$.
   Go to 3).

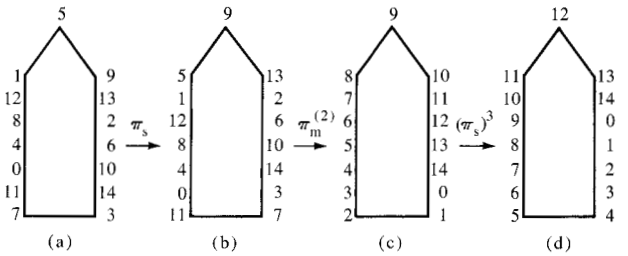

Figure 2 Sequence of transformations to access datum 12.

Remark Step 3) means that if the rightmost number a_1 is not zero, we apply enough π_s's to reduce it to zero. (Recall that each π_s subtracts 1 from the binary representation of $f(j)$.) Step 4) is to right rotate the resulting binary representation for d bits so that the previous number a_{i+1} now becomes the new a_i and the new a_α is zero.

Example Assume $n = 15$, $k = 4$, $d = 2$. In Fig. 2(a), the locations of data are listed before accessing. (Note that the sequence $\pi_s \pi_m^{(2)} \pi_s$ has been applied to the initial configuration to make this current configuration more complicated.) The memory map is $f(i) = (i \cdot 2^2 + 10)_{\bmod 15}$, i.e., $p = 1$, $q = 10$. Suppose datum 12 is to be accessed. Its location is $f(12) = 13$, whose binary representation is 1101. So there are two blocks each of length 2. The rightmost block is 01. We therefore apply one π_s to the memory, resulting in the configuration in Fig. 2(b). The location of datum 12 is now $f(12) = 12$, which is 1100. We now apply $\pi_m^{(2)}$, which changes the location of datum 12 to $(2^2 \cdot 12)_{\bmod 15} = 3$ as shown in Fig. 2(c). Therefore $f(12)$ is now 0011, which is a right-rotated 1100. We then apply 3 π_s's consecutively to bring datum 12 to the window (cell 0), as shown in Fig. 2(d). The memory map is now $f(i) = (i+3)_{\bmod 15}$, i.e., $p = 0$, $q = 3$.

It can be shown exactly as in [1] that the sequential accessing property (Theorem 2) holds for Algorithm 1' as well [4]. We are therefore concerned only with the accessing of a datum and analyze its performance for both the worst case and the average case.

To compute the number of π_s's and $\pi_m^{(d)}$'s generated by Algorithm 1' for accessing datum j, we represent the memory map of $f(j)$ in binary numbers as in step 1) of Algorithm 1'. If $a_i = 0$ for $i = 1, 2, \cdots, \alpha$, then both numbers are zero. Otherwise, the number of π_s's is equal to $\Sigma_{i=1}^{\alpha} a_i$ (see step 3) of Algorithm 1'), and the number of $\pi_m^{(d)}$'s is equal to $q - 1$, where $q = \max\{i | a_i > 0\}$ (see step 4) and the remark following Algorithm 1').

With this understanding, it is easy to see that to access any datum j, Algorithm 1' takes at most $S_{k,d} = 2^d \cdot \alpha - 2$ steps because a worst case occurs when datum j is located at cell $n-1$. The binary representation of its memory

Table 1 Steps needed to access a datum using Algorithm 1′.

	Worst case	*Average case*
d	$\frac{2^d}{d}\log_2(n+1)-2$	$\frac{2^d+1}{2d}\log_2(n+1)-\frac{2^d}{2^d-1}$
1	$2\log_2(n+1)-2$	$1.5\log_2(n+1)-2$
2	$2\log_2(n+1)-2$	$1.25\log_2(n+1)-1.333$
3	$2.667\log_2(n+1)-2$	$1.5\log_2(n+1)-1.143$
4	$4\log_2(n+1)-2$	$2.125\log_2(n+1)-1.067$

Table 2 Steps needed to access a datum using Algorithm 1″.

	Worst case	*Average case*
d	$\frac{2^{d-1}+1}{d}\log_2(n+1)-1$	$\frac{2^{d-2}+1}{d}\log_2(n+1)-\frac{2^d}{2^d-1}$
1	$2\log_2(n+1)-1$	$1.5\log_2(n+1)-2$
2	$1.5\log_2(n+1)-1$	$\log_2(n+1)-1.333$
3	$1.667\log_2(n+1)-1$	$\log_2(n+1)-1.143$
4	$2.25\log_2(n+1)-1$	$1.25\log_2(n+1)-1.067$

map $f(j)=n-1$ has all 1's except the rightmost bit. If we divide it into α blocks each of length d, then all blocks except the rightmost one have 1's in all d positions. The number of $\pi_m^{(d)}$'s generated is thus $\alpha-1$, and the number of π_s's generated is $(\alpha-1)(2^d-1)+(2^d-2)$, with a total of $2^d\cdot\alpha-2$.

To compute the average number of steps, i.e., the total number of $\pi_m^{(d)}$'s and π_s's generated by Algorithm 1′, we assume that all data are equally likely to be accessed. We show that to access any datum j, on the average, Algorithm 1′ takes

$$t_{k,d}\doteq\alpha-\frac{2^d}{2^d-1} \tag{10}$$

$\pi_m^{(d)}$'s and

$$v_{k,d}\doteq\frac{2^d-1}{2}\alpha \tag{11}$$

π_s's for large k.

To prove (10) and (11), recall that the memory map $f(j)$ ranges from 0 to $n-1$. Since we are interested in the case when k (hence n) is large, as a good approximation, we can assume that $f(j)$ ranges from 0 to n and each of these $n+1$ numbers is equally likely to occur. Such an assumption greatly simplifies the calculation because, if we represent a given $f(j)$ by a k-bit binary number, any bit has equal probability $1/2$ to be 0 or 1. Let t be the number of $\pi_m^{(d)}$'s generated for $f(j)$. Then the average value of t is given by

$$\begin{aligned}\sum_{i=0}^{\alpha-1} i\,P[t=i]&=\sum_{i=1}^{\alpha-1} i\,P[t=i]\\&=\sum_{i=0}^{\alpha-2}P[t>i]\\&=\sum_{i=0}^{\alpha-2}(1-P[t\le i]).\end{aligned}$$

To compute $P[t\le i]$, divide the binary representation of $f(j)$ into α blocks with representing numbers $a_1,\cdots,a_\alpha$ (from right to left) as before. Then for $0\le i\le\alpha-2$, $t\le i$ if and only if $a_{i+2}, a_{i+3},\cdots, a_\alpha$ are all zero. Thus $P[t\le i]=2^{-(\alpha-i-1)d}$, and the average value of t is $(\alpha-1)-[(1-2^{-k+d})/(2^d-1)]$. Equation (10) therefore follows.

Similarly, to compute the average value of the number of π_s's generated, it suffices to compute the average value of a_i, $1\le i\le\alpha$, and sum over i. But the average value of a_i is equal to $(2^d-1)/2$ for all i. Thus (11) follows.

Table 1 lists the number of steps needed to access a datum j for various values of d for both the worst case and the average case. For comparison, we express the formulae $2^d\cdot\alpha-2$, (10), and (11) in terms of $\log_2(n+1)$. Thus the best choice is $d=2$. Note that for large d the average value is approximately half of the worst case value.

An additional memory transformation

In this section, we introduce a new transformation

$$\pi_a(i)=(i+1)_{\bmod n}, \tag{12}$$

which is (4) in the reverse direction. The dynamic memory now has three kinds of transformations, namely, (4), (9), and (12). Algorithm 1′ can then be generalized to Algorithm 1″, which generates an accessing sequence of π_s's, π_a's, and $\pi_m^{(d)}$'s to fetch a datum j, and better performance results. In this memory, the memory address map is also given by (6′), and, in addition to (7) and (8′), we now have

$$\pi_a \text{ maps } \begin{cases} p \text{ into } p,\\ q \text{ into } (q+1)_{\bmod n}. \end{cases} \tag{13}$$

(Recall that $n=2^k-1$ and $k=\alpha d$.)

In step 3) of Algorithm 1′, if $a_1\neq 0$, a_1 π_s's are generated. With the introduction of π_a, we have the option of using π_a instead of π_s. Since $0\le a_1\le 2^d-1$, we use π_s's for $a_1<2^{d-1}$. But for $a_1>2^{d-1}$, we use π_a's to bring a_1 to 2^d so that a carry is generated and a_1 becomes zero. (The case $a_1=2^{d-1}$ needs special attention for later analysis to go through.) However, because of the presence of carries, we have to separate the case $2^{k-1}\le f(j)\le 2^k-1$ from the case $0\le f(j)\le 2^{k-1}-1$. In the

former case, the leftmost bit in the binary representation of $f(j)$ is always 1. A carry may cause overflow. Fortunately, this problem can be avoided if we consider the complementary representation of $f(j)$. For example, if $x = b_k \cdots b_1$ is the binary representation of $f(j)$, $(b_k = 1)$, then $\bar{x} = \bar{b}_k \cdots \bar{b}_1$ represents a number $\leq 2^{k-1} - 1$, where $\bar{b}_i = 1 - b_i$. Assuming that the accessing sequence generated for $\bar{x}$ is $S_1, S_2, \cdots, S_m$, then $\bar{S}_1, \bar{S}_2, \cdots, \bar{S}_m$ is an accessing sequence for the original x, i.e., $f(j)$, where

$$\bar{S}_i = \begin{cases} \pi_a \text{ if } S_i = \pi_s, \\ \pi_m^{(d)} \text{ if } S_i = \pi_m^{(d)}, \\ \pi_s \text{ if } S_i = \pi_a. \end{cases}$$

(See [4].)

We next give a detailed description of the algorithm and present analytic results concerning its performance. Note that the sequential accessing property is also preserved in this memory structure.

Algorithm 1″ Given p, q (the memory map) and j (the datum to be fetched), this algorithm generates an accessing sequence of π_s's, π_a's, and $\pi_m^{(d)}$'s. (Updating of p, q is omitted for simplicity.)

Assume $0 \leq f(j) \leq 2^{k-1} - 1$; otherwise consider complementary representation.

1. Let a be the k-bit binary number representing $f(j)$. Then the leftmost bit of a is always zero. Divide a into α blocks of d bits each. Number the blocks $1, 2, \cdots, \alpha$ from right to left. Let the numbers represented by the blocks be $a_1, a_2, \cdots, a_\alpha$, respectively. Then $0 \leq a_i \leq 2^d - 1$ for all $i < \alpha$, and $0 \leq a_\alpha \leq 2^{d-1} - 1$.
2. $\beta \leftarrow 0$. (This is the carry parameter. If $\beta = 1$, the current value of a_1 is the sum of a carry from the right and its old value.)
3. If all $a_i = 0$, stop.
4. If $a_1 = 0$, go to 5). Otherwise we have the following cases:
   a. For $0 \leq a_1 < 2^{d-1}$, apply a_1 π_s's to the memory. $a_1 \leftarrow 0$, $\beta \leftarrow 0$.
   b. For $2^{d-1} < a_1 \leq 2^d - 1$, apply $(2^d - a_1)$ π_a's to the memory. $a_1 \leftarrow 0$, $\beta \leftarrow 1$.
   c. For $a_1 = 2^{d-1}$ and $\beta = 0$, apply a_1 π_a's to the memory. $a_1 \leftarrow 0$, $\beta \leftarrow 1$.
   d. For $a_1 = 2^{d-1}$ and $\beta = 1$, apply a_1 π_s's to the memory. $a_1 \leftarrow 0$, $\beta \leftarrow 0$.
   e. For $a_1 = 2^d$, $a_1 \leftarrow 0$, $\beta \leftarrow 1$.
   Go to 3).
5. Generate $\pi_m^{(d)}$.
   $a_1 \leftarrow a_2 + \beta$. $a_i \leftarrow a_{i+1}$, for $i = 2, \cdots, \alpha - 1$. $a_\alpha \leftarrow 0$.
   Go to 4).

Note that when $a_1 = 2^{d-1}$, we can choose either π_a or π_s. For simplicity of analysis, we use the present rule, i.e., if $a_1 = 2^{d-1}$ is not the result of a carry from the right, then π_a's are used and a carry is generated from this block. On the other hand, if $a_1 = 2^{d-1}$ is the result of a previous carry, then π_s's are used and no carry is generated. This makes the probability for a block to generate a carry 1/2. (See [4].)

After an analysis similar to but considerably lengthier than before [4], we find that to access any datum j, Algorithm 1″ takes at most $W_{k,d}$ steps, where

$$W_{k,d} = \begin{cases} (2^{d-1} + 1)\alpha - 2, \text{ for } \alpha \text{ odd}, & (14) \\ (2^{d-1} + 1)\alpha - 1, \text{ for } \alpha \text{ even}. & (14') \end{cases}$$

For the average case, to access a datum j, the total number of π_s's and π_a's generated by Algorithm 1″ is $u_{k,d}$, where

$$u_{k,d} \doteq 2^{d-2} \cdot \alpha \qquad (15)$$

for large k. The total number of $\pi_m^{(d)}$'s generated by Algorithm 1″ is $v_{k,d}$, where

$$v_{k,d} \doteq \alpha - \frac{2^d}{2^d - 1} \qquad (16)$$

for large k.

Remark Our computation [4] shows that $v_{k,d}$ and $t_{k,d}$ (Eq. (10)) have not only the same approximation but also the exact formulae. This means that the effect of the carries on the number of $\pi_m^{(d)}$'s is being averaged out.

Table 2 lists the number of steps needed to access a datum j for various values of d for both the worst case and the average case. (See (14′), (15), and (16).)

Again, $d = 2$ gives the best performance. For d large, the average value is approximately half of the worst case value. Also note that for large d, both the worst case value and the average case value for Algorithm 1″ are approximately half of those for Algorithm 1′.

Finally, it should be pointed out that the improvements of this memory structure (for $d = 2$) are about 25 and 33 percent over the original memory proposed in [1] for the worst case and the average case, respectively.

Generalization to arbitrary radix

As in [1], we can generalize our memory structure using an arbitrary radix. Let $n = r^k - 1$ and $k = \alpha d$, where r, α, d are positive integers and $r \geq 2$. The counterparts of the transformations in the section on "A class of memory transformations" are π_s and

$$\pi_{m,r}^{(d)}(i) = (i \cdot r^{(\alpha-1)d})_{\bmod n}, \qquad (17)$$

for $i = 0, 1, \cdots, n - 1$. Those of the transformations in the section on "An additional memory transformation" are π_s, π_a, and $\pi_{m,r}^{(d)}$.

Note that if we represent i as a k-digit base r number, then the effect of $\pi_{m,r}^{(d)}$ is to right rotate it for a block of d digits.

Table 3 Worst case number of steps required using Algorithm $1'_r$.

$$\frac{r^d}{d \log_2 r} \log_2(n+1) - 2$$

d \ r	2	3	4	5
1	$2 \log_2(n+1) - 2$	$1.893 \log_2(n+1) - 2$	$2 \log_2(n+1) - 2$	$2.153 \log_2(n+1) - 2$
2	$2 \log_2(n+1) - 2$	$2.839 \log_2(n+1) - 2$	$4 \log_2(n+1) - 2$	$5.383 \log_2(n+1) - 2$
3	$2.667 \log_2(n+1) - 2$	$5.678 \log_2(n+1) - 2$	$10.667 \log_2(n+1) - 2$	$17.945 \log_2(n+1) - 2$
4	$4 \log_2(n+1) - 2$	$12.776 \log_2(n+1) - 2$	$32 \log_2(n+1) - 2$	$67.293 \log_2(n+1) - 2$

Table 4 Average number of steps required using Algorithm $1'_r$.

$$\frac{r^d+1}{2d \log_2 r} \log_2(n+1) - \frac{r^d}{r^d-1}$$

d \ r	2	3	4	5
1	$1.5 \log_2(n+1) - 2$	$1.262 \log_2(n+1) - 1.5$	$1.25 \log_2(n+1) - 1.333$	$1.292 \log_2(n+1) - 1.25$
2	$1.25 \log_2(n+1) - 1.333$	$1.577 \log_2(n+1) - 1.125$	$2.125 \log_2(n+1) - 1.067$	$2.799 \log_2(n+1) - 1.042$
3	$1.5 \log_2(n+1) - 1.143$	$2.944 \log_2(n+1) - 1.038$	$5.417 \log_2(n+1) - 1.016$	$9.044 \log_2(n+1) - 1.008$
4	$2.125 \log_2(n+1) - 1.067$	$6.467 \log_2(n+1) - 1.013$	$16.063 \log_2(n+1) - 1.004$	$33.7 \log_2(n+1) - 1.002$

Algorithm $1'$ can now be generalized to Algorithm $1'_r$ in the following manner. In step 1) of Algorithm $1'$, we now represent $f(j)$ as a k-digit base r number instead of a binary number. In step 4), we use $\pi^{(d)}_{m,r}$ instead of $\pi^{(d)}_m$. A similar analysis of the algorithm can be carried out [4].

To access any datum j, Algorithm $1'_r$ takes at most $S_{k,d,r}$ steps, where

$$S_{k,d,r} = r^d \cdot \alpha - 2. \tag{18}$$

If $t_{k,d,r}$ is the average number of $\pi^{(d)}_{m,r}$'s generated by Algorithm $1'_r$, then

$$t_{k,d,r} \doteq \alpha - \frac{r^d}{r^d-1} \tag{19}$$

for large k. If $c_{k,d,r}$ is the average number of π_s's, then

$$c_{k,d,r} \doteq \frac{r^d-1}{2} \alpha \tag{20}$$

for large k.

We tabulate the results of (18), (19), and (20) in Tables 3 and 4.

Note that both formulae are invariant under the transformation $r \to r^2$, $d \to d/2$. This explains why the entries for $d=2$, $r=2$ and for $d=1$, $r=4$ are identical. This is because a change of radix of the form $d \to d/2$ is equivalent to taking twice as many digits at a time, which is exactly balanced by halving the number of digits in the higher radix.

Referring to the tables, $d=1$, $r=3$ is the best for the worst case and $d=2$, $r=2$ (or $d=1$, $r=4$) is the best for the average case. Note that for large d and fixed r, the average value is half of the worst case value.

Algorithm $1''$ can be generalized to Algorithm $1''_r$ for arbitrary radices in a similar fashion except for handling r even and r odd. For r even, as in Algorithm $1''$, if $a_1 < r^d/2$, apply a_1 π_s's to turn it to zero. If $a_1 > r^d/2$, apply $(r^d - a_1)$ π_a's to bring it up to r^d and a carry is generated. When $a_1 = r^d/2$, we apply either a_1 π_s's or a_1 π_a's, depending on the existence of a carry from the right. If π_a's are applied, a carry is generated. This forces the probability for a block to generate a carry to be $1/2$ as in the binary case. For r odd, r^d is also odd. Thus, the probability for a block to generate a carry can never be $1/2$, and we do not make a special case for $a_1 = (r^d-1)/2$, i.e., if $a_1 \leq (r^d-1)/2$, apply a_1 π_s's, and if $a_1 > (r^d-1)/2$, apply $(r^d - a_1)$ π_a's.

By reasoning similar to that used before, one can show [4] that for r even, to access any datum j, Algorithm $1''_r$ takes at most $W_{k,d,r}$ steps, where

$$W_{k,d,r} = \begin{cases} \dfrac{r^d+2}{2} \alpha - 2 & \text{for } \alpha \text{ odd}, \quad (21) \\ \dfrac{r^d+2}{2} \alpha - 1 & \text{for } \alpha \text{ even}. \quad (22) \end{cases}$$

If $u_{k,d,r}$ is the average total number of π_s's and π_a's generated by Algorithm $1''_r$, then

Table 5 Worst case results.

	r even: $\frac{r^d+2}{2d\log_2 r}$			r odd: $\frac{r^d+1}{2d\log_2 r}$				
d \ r	2	3	4	5	6	7	8	9
1	2	1.262	1.5	1.292	1.547	1.425	1.667	1.577
2	1.5	1.577	2.25	2.799	3.675	4.453	5.5	6.467
3	1.667	2.944	5.5	9.044	14.055	20.425	28.556	38.393
4	2.25	6.467	16.125	33.7	62.766	106.965	170.75	258.836

Table 6 Average results.

	$\frac{r^d+4}{4d\log_2 r}$							
d \ r	2	3	4	5	6	7	8	9
1	1.5	1.104	1	0.969	0.967	0.980	1	1.025
2	1	1.025	1.25	1.561	1.934	2.360	2.833	3.352
3	1	1.630	2.833	4.630	7.092	10.302	14.333	19.275
4	1.25	3.352	8.125	16.931	31.431	53.549	85.417	129.477

$$u_{k,d,r} \doteq \frac{r^d}{4}\alpha \tag{23}$$

for large k. The average number of $\pi_{m,r}^{(d)}$'s generated is

$$v_{k,d,r} \doteq \alpha - \frac{r^d}{r^d-1} \tag{24}$$

for large k.

When r is odd, to access any datum j, Algorithm $1_r''$ takes at most $\bar{W}_{k,d,r}$ steps, where

$$\bar{W}_{k,d,r} = \frac{r^d+1}{2}\alpha - 1. \tag{25}$$

The average total number of π_s's and π_a's is

$$\bar{u}_{k,d,r} \doteq \frac{r^d}{4}\alpha \tag{26}$$

for large k. The average number of $\pi_{m,r}^{(d)}$'s is

$$\bar{v}_{k,d,r} \doteq \alpha - \frac{r^d}{r^d-1}, \tag{27}$$

for large k, i.e., the same as when r is even.

We tabulate the worst case results of (22) and (25) in Table 5. For r even, it is $[(r^d+2)/(2d\log_2 r)]\log_2(n+1)-1$, and for r odd, it is $[(r^d+1)/(2d\log_2 r)]\log_2(n+1)-1$. Only the coefficients of $\log_2(n+1)$ are listed.

For the average case, we only consider the term $[(r^d+4)/(4d\log_2 r)]\log_2(n+1)$. (See (23), (24), (26), and (27).) We tabulate the values of $(r^d+4)/4d\log_2 r$ in Table 6. Note that these formulae are also invariant under the transformations $r \to r^2$, $d \to d/2$. For the worst case, $r=3$, $d=1$ gives the best performance. For the average case, $r=6$, $d=1$ gives the best performance. For $d=1$, $r=4, 5, 7, 8$ and $r=2$, $d=2, 3$ performance is comparable. Note that for large d and fixed r, the average value is half of the worst case value. Also for large d, both the worst case value and the average case value for Algorithm $1_r''$ are approximately half of those for Algorithm $1_r'$.

Conclusions

In this paper, we extend the results in [1] in two directions. By introducing a parameter d and considering right rotations instead of left rotations, we extend the shuffle transformation to a family of transformations that might be termed block shuffles. The algorithms proposed in [1] are also extended accordingly. Better performance results when the appropriate parameter is chosen. By adding a new transformation, namely, the reverse cyclic shift, a new memory structure with still better performance results.

Finally, all memory structures and results are generalized to the case of arbitrary radix.

It should be pointed out that the improvements of [1] by Stone [5] are also applicable here.

References

1. A. V. Aho and J. D. Ullman, "Dynamic Memories with Rapid Random and Sequential Access," *IEEE Trans. Comput.* **C-23,** 272 (1974).
2. H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Comput.* **C-20,** 153 (1971).
3. H. S. Stone, "Dynamic Memories with Enhanced Data Access," *IEEE Trans. Comput.* **C-21,** 359 (1972).
4. C. K. Wong and D. T. Tang, "Dynamic Memories with Faster Random and Sequential Access," *Research Report RC 5682,* IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1975.
5. H. S. Stone, "Dynamic Memories with Fast Random and Sequential Access," *IEEE Trans. Comput.* **C-24,** 1167 (1975).
6. T. Lang, "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network," Stanford University, CA; available as reprint *R74-19* from IEEE Computer Society Repository.
7. T. Lang, "Performing the Perfect Shuffle in an Array Computer," Stanford University, CA; available as reprint *R74-20* from IEEE Computer Society Repository.
8. T. Lang and H. S. Stone, "A Shuffle-Exchange Network with Simplified Control," *IEEE Trans. Comput.* **C-25,** 55 (1976).
9. D. H. Lawrie, "Access Requirements and Design of Primary Memory for Array Processors," University of Illinois, Urbana; available as reprint *R74-30* from IEEE Computer Society Repository.
10. D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. Comput.* **C-24,** 1145 (1975).
11. J. Lenfant, "Fast Random and Sequential Access to Dynamic Memories of any Size," *Research Report No. 34,* Université de Rennes, Département de Mathématique et Informatique, 1976.
12. W. F. Beausoleil, D. T. Brown, and B. E. Phelps, "Magnetic Bubble Memory Organization," *IBM J. Res. Develop.* **16,** 587 (1972).
13. P. I. Bonyhard and T. J. Nelson, "Dynamic Data Relocation in Bubble Memories," *Bell Syst. Tech. J.* **52,** 307 (1973).
14. C. Tung, T. C. Chen, and H. Chang, "A Bubble Ladder Structure for Information Processing," *IEEE Trans. Magnet.* **MAG-11,** 1163 (1975).
15. C. K. Wong and D. Coppersmith, "The Generation of Permutations in Magnetic Bubble Memories," *IEEE Trans. Comput.* **C-25,** 254 (1976).

Received September 6, 1976

The authors are located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.