Realization of a Processor with Virtual Tapes and Its Evaluation

KOZO ITANO* and EIICHI GOTO**

Architecture of a processor equipped with a "virtual tape" memory is presented, and programming examples and their performance evaluations are given. The physical entity of the virtual tape consists of data blocks on auxiliary memory buffered in main memory by employing a look-ahead swapping technique. Logically, a programmer can use a virtual tape as fast and unlimited tape memory. The virtual tape greatly simplified programs involving stacks, queues and I/O buffers. Matrix operation and fast Fourier transform program are shown to run more efficiently on a virtual tape machine than on a conventional paged virtual memory.

1. Introduction

In all virtual memory schemes, high data locality is required for high efficiency. The data structure which has the best locality is evidently a linear list accessed in a purely sequential mode. We shall use the term "virtual tape" or its abbreviation "V-tape" for this data structure to indicate the fact it is the best data structure for virtual memory. Historically, Turing used this data structure in his hypothetical machine [1]. Turing used the term "tape" for a purely abstract data structure. In contemporary computers, however, "tape" usually means "physical (magnetic or paper) tapes". The article "virtual" would clearly discriminate abstract structures from physical tapes.

Because of the high data locality, more efficient memory management algorithms can be used for V-tapes than for more general data structures with less locality in a virtual memory environment. As in a Turing machine, we place a "head" on the presently accessed data cell of a V-tape. If the data area containing the data cell under the head and its neighbouring cells are residing in a buffer area in the high speed memory, the tape operations of the type as used in a Turing machine can be executed at high speed over a V-tape. As long as the head moves sequentially, missing page faults can be reduced by applying a look-ahead swapping principle instead of the on-demand swapping principle applied to data structures more general than V-tapes.

However, because the Turing type operations are too restrictive for practical programming, we added some other operations to be explained in the following and we built a V-tape machine which executes these tape operations and manages the memory hierarchy automatically using the look-ahead swapping principle.

The V-tape machine is composed of three handlers (sub-machines) as shown in Fig. 1. One of them is BCPL INTCODE [2] handler which is a base processor of the machine; its instruction set is BCPL INTCODE as the name indicates. Another machine is the V-tape handler which is the processor for handling the V-tape. In order to evaluate the performance of the V-tape machine, we also built a virtual memory handler which performs the memory management by using damand paging and LRU (Least Recently Used) replacement policies. All three handlers are microprogrammed on the processor which we built, and they are connected to memory and I/O as shown in Fig. 2.

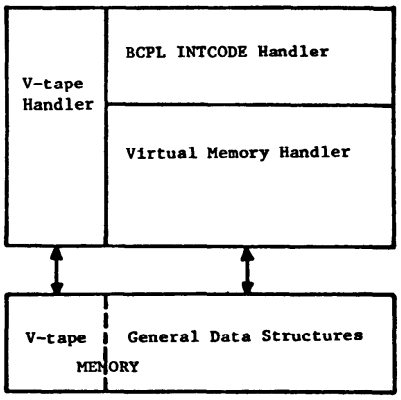

Fig. 1 The V-tape machine.

The V-tape concept was originally suggested by E. Goto and was first reported by L. Mateev and present authors [3]. Computational complexity of machines (automata) equipped with multihead V-tapes has been treated by M. Sassa and E. Goto [4]. Discussions and results obtained by the end of 1974 was summarized by L. Mateev [5]. Refined considerations and implementational details for the hardware such as hysteresis, swap request queue and buffer memory management were made and reported by K. Itano [6, 7] in great detail.

In this paper, the basic operational principles of the hardware are summarized in section 3 without going

*Institute of Information Sciences and Electronics, University of Tsukuba, Sakura-mura, Niihari-gun, Ibaraki 300-31, Japan.
**Department of Information Science, University of Tokyo, Hongo, Bunkyo-ku, Tokyo 113, Japan.

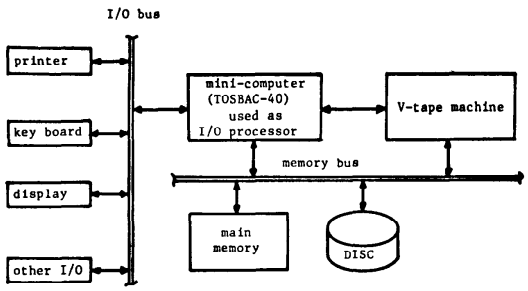

Fig. 2 Hardware configuration of the processor.

into details of the hardware design. The management of the hierarchy of the memory (look-ahead page swapping, buffer allocation, etc) is handled by the hardware and it is transparent from the programmer at the machine code level. The basic instruction set of the V-tape machine is explained in section 2, and basic operational principles in section 3. In section 4, applications of V-tapes to matrix manipulation, FFT and sorting are disclosed and performance evaluation of V-tapes are made through a comparative studies with conventional on-demand paging machine.

2. Basic Instructions for the Virtual Tapes and Their Use

2.1 Turing Machine

One of the most obvious applications of the V-tape is the realization of Turing machines. For the purpose of implementing multitape Turing machines by using V-tapes, the V-tape system should provide a facility which enables us to create many V-tapes with a data access time comparable to the main memory, even though the physical system actually has a relatively small main memory. Since the V-tape consists of sequentially arranged data cells, it can be accessed only relatively and sequentially. For the access of the V-tape, a head is located on a data cell in the V-tape, and it can move rightwards or leftwards to the next cell over the V-tape. Logically, a V-tape is completely symmetric in both directions. For convenience, however, we call one ending of the V-tape "BOT" (Beginning Of the Tape) and the other ending of the V-tape "EOT" (Ending Of the Tape). In contrast to magnetic tape devices, the head over V-tapes can move in both directions equally. For convenience, we call the direction towards EOT "forward" or "right"; towards BOT "backward" or "left". The V-tapes are automatically (transparently from programmers) extended. Namely, when a head enters a BOT (EOT respectively) the tape is extended to left (right) automatically with all blanks (for Turing machine) or zeros (for numerical applications) being supplied on the extended part of the tape. In other words, a storage for V-tape is allocated and the tape is extended "on-demand" (a dynamic storage allocation principle for reducing storage requirement).

Most of the physical entity of a V-tape resides in the auxiliary memory such as magnetic drums or magnetic discs, and only a part of the data area containing the data cell under a head and data cells in the neighborhood of a head are read out into the buffer in the main memory. In order to manage V-tapes, the physical entity of the V-tape is divided into fixed sized pages which are dynamically allocated in buffers in the main memory or in the auxiliary memory. A head is always made to point at the data cell in a buffer page which resides in the main memory. This scheme enables us to put many heads on the V-tape, as far as the buffer area in the main memory is available. Further, two or more heads on the V-tape can be located on the same position, and a head can even pass through another head. In the physical tapes such as magnetic tapes, however, it would be very difficult to perform such operations.

For actual programming of the Turing machine, we need instructions to handle the V-tape. First, we introduce the following four instructions: CREATETAPE, DELETETAPE, CREATEHEAD, DELETEHEAD.

tape number←CREATETAPE (*erase mode*):
This instruction creates a new V-tape with *erase mode* which is explained later, and initializes the system table and registers for the new V-tape. The resultant value of this instruction is the "*tape number*" identifying the V-tape just created.

DELETETAPE (*tape number*, $\cdots$):
This instruction deletes the V-tape(s) specified by the "*tape number(s)*" from the V-tape system. The pages belonging to the V-tape are returned to the free storage.

head number←CREATEHEAD (*tape number*):
This instruction creates a new head, initializes the tape system table and registers, and PLACEs the head on the BOT of the tape specified by "*tape number*". The resultant value of this instruction is the "*head number*" identifying the head just created.

DELETEHEAD (*head number*, $\cdots$):
This instruction deletes the head(s) specified by the "*head number(s)*" from the system. The page allocated in the buffer in the main memory for this head is released. Use of a garbage collection for "delete" operations would be more convenient for the general users. In this case, only the system programmer who write the garbage collector will use delete operations.

Further, in order to implement a Turing machine, we need the following head instructions:

VT$\alpha\beta\gamma$ (*head number*, *data*)

where α, β and γ are one of the letters R, W, F, B, and E or a void, whose actions are as follows:

R: Read data from the data cell referred by the head.
W: Write data into the data cell referred by the head.
F: Move the head forwards by one cell.
B: Move the head backwards by one cell.
E: Erase the data cell referred by the head.

In case of VT$\alpha\beta\gamma$ operation, the action takes place in

the order α, β, γ. For example, VTRF (*head number*, *data*) specifies an operation that read data and move forwards (γ is a void).

Finally, the size of the physical data cell depends upon "word" and "byte" length of the target machine which are 16 and 8 bits in the actual system. Therefore, if a user wants to extend the data size, he should solve this by programming.

2.2 Stacks, Queues, Deques and Multiheaded Queues

Stacks and queues are well known special cases of sequentially accessed linear list, and are important and useful data structures. Stacks are used in parameter tracking in recursive algorithms, and they are also used for subroutine linkage and priority interrupts. Queues are used in I/O buffering. Moreover stacks and queues are used for list compacting algorithms [8].

Stacks and queues are easily implemented by using V-tapes. In case of a stack, push down is made by performing the VTWF instruction, and pop up by the VTBR instruction. In order to implement queues and deques, we need two heads on a V-tape. In case of a queue, one head is used to write data into the list by the VTWF instruction and the other to read by the VTRF instruction. In case of a deque [9], both heads are used to write and to read the data; in one head by VTRF and VTBW instructions and in the other head by VTBR and VTWF instructions. In recursive language systems with garbage collection schemes such as LISP, it is more precise to consider its system stack as a more complex V-tape structure (a "scannable stack" [4, 5]) with two heads instead of one, because the content of the stack must be scanned and read during the garbage collection.

By the use of a V-tape, a more complicated queue (a "multiheaded queue") with two or more read heads can be realized easily. All we have to do is to create heads needed and to locate them on the V-tape. Multiheaded queue is useful for multiple buffering operations. For example, Fig. 3 shows the diagram to perform the buffering of two output devices with a multiheaded queue using a V-tape with three heads on it. In this diagram, a V-tape T is created as a multiheaded queue; two read heads HA, HB are created on the V-tape T; they are connected to the output device LP (high speed printer) and PTP (high speed punch) respectively. We create another head HC on the V-tape T. When we write data by this head HC, the data buffering is performed automatically using the V-tape T.

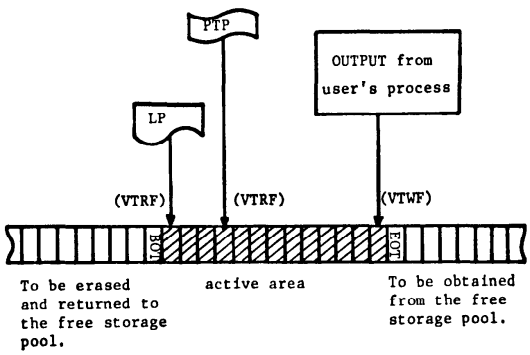

Fig. 3 Multi-headed queue for I/O buffering.

2.3 Erasing of Virtual Tapes

In case of a stack, cells left (backward) to the head (stack top) only contains active information. Hence, the cells and their buffer resources right (forward) to the head may be erased and returned to the pool of system buffer resources. We could use a VTBRE (move Backwards, Read, and Erase) operation instead of a VTBR operation for pop up, and in case of queue use a VTREF (Read, Erase, and move Forwards) operation instead of a VTRF operation. However, when multiheaded queue is to be erased, a head with erase capability such as VTBRE and VTREF instructions would not work well. If there are two or more heads for reading the queue, it is rather difficult to determine the one which should perform erase, because a reading head may pass another reading head. This problem can be resolved by imparting the erasing action to the BOT and EOT (Beginning and Ending Of the Tape) of a V-tape, rather than to the VT-instructions on heads. Tape ends with erasing action are called EBOT (Erasing BOT) and EEOT (Erasing EOT). In case EBOT (EEOT) is used instead of BOT (EOT), EBOT (EEOT) moves forwards (backwards) erasing the data cells until it reaches leftmost (rightmost) head or "mark" (to be defined later) on the V-tape. The memory resources corresponding to erased part of the V-tape are returned to the systems free stoarge pool.

2.4 PLACE

From the theoretical point of view, a Turing machine with a tape and a head is capable of programming all computable calculations. However, when it comes to practical programming, the feature that a head can only move either rightwards and leftwards by one data cell would be too restrictive in many cases. Therefore, the introduction of a few more devices would be desirable for the sake of speed and flexibility.

It is desirable to provide a mechanism which is capable of preserving the information of location on the V-tape, and to move a head directly to that position. We introduce a "mark" for this purpose. A mark is used to denote a special location on the V-tape. We also introduce a function, "PLACE", which enables us to move a head directly to the location refered by another head or mark. BOT and EOT of the V-tapes can be regarded as special marks by the V-tape system. The following instructions are installed for the above processing:

mark number←CREATEMARK ( ):

This instruction creates a new mark, reserving an

entity in a table (the mark table) to preserve the information of location on the V-tape. The result of this instruction is the *mark number* identifying the mark just created.

DELETEMARK (*mark number*):
This instruction deletes the mark specified by the *mark number* from the system (actually it is deleted from the mark table).

PLACE (*h/m number 1*, *h/m number 2*):
In this instruction, *h/m number* means either *head number* or *mark number*. This instruction moves the head or the mark specified by the *h/m number 1* to the position on which the head or the mark specified by the *h/m number 2* is located.

As a matter of fact, it is possible to use a head instead of a mark, but the introduction of marks is justified since its creation or PLACEment requires less system resources. Although a mark represents the same concept as a head logically, the difference between them consists in that a marked data cell may not be in the buffer in the main memory while a data cell under a head is always kept in the buffer. Therefore, we have to use marks and heads properly, considering the balances between the size of the buffer area and the frequency of the usage of heads and/or marks.

2.5 Random Access Extension and Pseudo-heads

Although the reference patterns of the V-tape disclosed so far are purely sequential accesses, extension in capability to random access in local range is useful. Two types of random access operations are treated in this section. All these operations include the letter X (eXtension).

We introduce a function, "MOVE", which moves a head more than one data cells over the V-tape at one time. MOVE is useful as a programming convenience. MOVE is slower than PLACE when the stride of the MOVEment is larger relative to the size of the page. MOVE is performed by the instructions VTFX and VTBX:

VTFX (*head number*, x) = VTBX (*head number*, $-$x):
These instructions move the head specified by the *head number* x data cells forwards or backwards relatively from the current position. In case of the instruction VTFX, if x is positive the head moves forwards and if negative, backwards. In case of the instruction VTBX, the direction of movement is reversed.

Also we introduce "pseudo-head" instructions. We perform the access by specifying relative distance x from the current position of the head as follows:

VTRX (*head number*, *data*, x)

VTWX (*head number*, *data*, x)

In this case, in order to access the data cell a "pseudo-head" is created on the V-tape by the system automatically. The system treats pseudo-heads similarly to heads except in that they can not be handled (created, moved, etc.) directly by the programmer. Hence, "pseudo" is prefixed.

Since VTFX, VTRX and VTWX instructions have the same aspect, they are called X-instructions. If "x" is larger relative to the size of the page p, |x/p| swaps are needed in case pointers are held in the pages. When pointers are held separately in the high speed memory (link table), at most one swap is enough to execute the instructions. For resource utilization, it would be reasonably economical to keep the pointers in the high speed memory. In the actual system, four bytes are used for pointers per page. Therefore, they occupy only $1/128 \sim 1/512$ of the whole data area for V-tapes in case the page size is 512 bytes $\sim$2048 bytes. Under these situation, we need |x/p| accesses to the link table for every execution of a X-instruction. This can be improved by applying a hash technique (software or hardware [10]) for page address mapping. Therefore, even if a programmer uses X-instructions in the worst manner, the execution speed of the instruction would only decrease to that of paged virtual memory systems.

3. Basic Operational Principles of Virtual Tape Machine

3.1 Management of the Auxiliary Memory

As mentioned earlier, the physical entity of the V-tapes consists of fixed sized pages. Each page has two boundaries; a top boundary and a bottom boundary. A bottom boundary of a page is connected to a top boundary of the next page, except for the two terminal page boundaries. We managed unused pages (i.e. free storage) by using a bit table, in which the disc map is held. Erasing of V-tapes is performed independently by a dedicated software processor as a job of the lowest priority. The page pointers for chaining pages in use are held separately in a table in the main memory.

3.2 Management of Buffers

When a head enters a BOT (EOT respectiely) on the top (bottom) boundary of the page, a new page is attached to the tape automatically in order to extend the V-tape. In case erasing action is performed by a EBOT (EEOT respectively) on the bottom (top) boundary of the page, the page is returned to the free storage pool automatically.

Since more than one heads may reside in the same page, different buffers should not be allocated to such heads for consistency. In other words, a buffer is to be allocated to a page rather than to a head.

A page is called "headed" if there is a head on it. A page in the neighborhood of a headed page and is called "near-headed". Otherwise, a page is called "unheaded". An "unheaded" page is called "marked" if there is a mark on it, otherwise the page is called "unmarked".

Since all pages cannot reside in the buffer because of the main memory limitation, some kind of strategy and

algorithm are necessary to decide which page should be kept in the buffer. It would be a reasonable strategy for the V-tape system to keep "headed" pages in the buffer with the highest priority and to keep "near-headed" pages in the buffer with the second highest priority. However, in the actual system, both "headed" and "near-headed" pages are kept in the buffer with the same and highest priority for the sake of simplicity. When a new buffer area is requested by a "near-headed" page, an "unheaded" page is swapped out to the auxiliary memory.

In a practical V-tape system, the available buffer in the main memory is limited. Therefore, the number of the head which a programmer can create is restricted. In case a head moves only in one direction, however, we can reduce the number of the buffer page needed by assigning a "head motion mode" to each head [4, 7].

4. Applications and Performance Evaluations

4.0 Programming Language

The actual application programs were written in a variant of BCPL. However, in order to improve readability, the programs to follow will be given in FORTRAN.

4.1 Matrix Operations

(1) Storage Scheme for Matrices

In case of V-tape, the most natural way to keep a matrix on the V-tape is to use the packed column storage scheme [11].

(2) Matrix Multiplication

Let A, B and C be N by N matrices. The matrix multiplication of A and B into C is defined as the following formula: $C_{ij}=\Sigma_k A_{ik} \times B_{kj}$. Then, the obvious program to perform this is:

```
program M1
    DO 10 J=1, N
    DO 10 I=1, N
    C(I, J)=0
    DO 10 K=1, N
10  C(I, J)=C(I, J)+A(I, K)*B(K, J)
```

However, this program cannot be performed effectively on the hierarchical memory [11]. Therefore, we rewrite this by the programming technique called "ordering nested loop" [11]. The improved program M2 is shown below.

```
program M2
    DO 20 J=1, N
    DO 10 I=1, N
10  C(I, J)=0
    DO 20 K=1, N
    DO 20 I=1, N
20  C(I, J)=C(I, J)+A(I, K)*B(K, J)
```

The advantage of this algorithm is reported by J. L. Elshoff [11]. This algorithm M2 can be transformed into the V-tape algorithm VTM2 which uses three V-tapes and five heads.

```
program VTM2
Input matrices A, B on V-tapes TA, TB.
Output matrices C on V-tape TC.
    HA=CREATEHEAD (TA)
    HA2=CREATEHEAD (TA)
    HB=CREATEHEAD (TB)
    HC=CREATEHEAD (TC)
    HC2=CREATEHEAD (TC)
    DO 10 J=1, N
    CALL PLACE (HA, HA2)
    CALL PLACE (HC2, HC)
    DO 20 K=1, N
20  CALL VTWF (HC, 0)
    DO 10 I=1, N
    CALL PLACE (HC, HC2)
    DO 30 K=1, N
      // this loop is coded in micro-program.
      CALL VTRF (HA, DA)
      CALL VTR (HB, DB)
      CALL VTR (HC, DC)
      DA=DA*DB+DC
30    CALL VTWF (HC, DA)
10  CALL VTF (HB)
    CALL DELETEHEAD (HA)
    CALL DELETEHEAD (HA2)
    CALL DELETEHEAD (HB)
    CALL DELETEHEAD (HC)
    CALL DELETEHEAD (HC2)
```

(3) Gaussian Elimination

Let us consider a system for solving linear equations, $Ax=b$. When the matrix A is dense, direct elimination methods are almost always the most efficient. Gaussian elimination is one of the best known and important direct methods for solving linear equations. Let $Ax=b$ be the equation to be solved, where A is a square matrix of order N, x and b are vectors of order N respectively. The programs: DECOMP (triangulation program) and SOLVE (back substitution program) for Gaussian elimination have been reported by G. Forsythe [12], and we have translated them into the programs for V-tapes.

(4) Two-dimensional FFT Algorithm

The essence of the program of the algorithm of one dimensional FFT is in the decomposition of original transform of size $N=2^n$ into n steps of N/2 transforms of size 2. This idea was developed by Cooley and Tukey [13]. The radix two transform algorithm on the sequential auxiliary memory was reported by R. C. Singleton [14]. FFT algorithms based on the V-tape concept and programs written for the V-tape machine are disclosed in [7] by the author.

(5) Performance Measurements

A 256 by 256 of 16 bit integer matrix needs 128 K bytes of storage, 384 K bytes for three such matrices, and 512 K bytes for 32 bits floating point data matrix,

which seems to be a reasonable data size to be handled by V-tapes.

(i) Matrix Multiplication

Measurements made in case of multiplication of 16 bit integer matrices of order 256 are shown in Fig. 4. These measurements are made with both programs M2 and VTM2. The V-tape system reduced the elapsed time about 75% in comparison with the time that elapsed in the paged virtual memory system.

(ii) Gaussian Elimination

A 256 dimension of linear equation was solved by using Gaussian elimination. The measurements made are shown in Fig. 4. In case of DECOMP, we reduced the elapsed time to 64~83% in comparison with the time that elapsed in the paged virtual memory system. In case of SOLVE, we reduced the elapsed time to 50~67%.

(iii) FFT

The Fig. 4 show that the column-wise transform of FFT requires roughly the same amount of time of swaps in both memory systems. Unnecessary swaps of row-wise transform of the V-tape processor is due to PLACE which is performed as far as channel capacity permits. Totally, we have reduced the elapsed time of processing to 65~67% of the time consumed in the demand paged virtual memory.

4.2 Sorting in a Virtual Tape System

(1) Sorting Algorithms on V-tapes

Let us consider the process of external sorting, for which "the balanced two-way merge" [15] is the simplest method. While 4 tapes (and 4 heads) are required in case of magnetic tapes, we can perform the balanced two-way merge sorting with 2 V-tapes and 3 heads; one V-tape for input with two reading heads and the other V-tape for output of merge with one writing head. At the beginning of each merge, two reading heads are to be PLACEd at the BOT of the input tape, and one of the two heads is to be MOVEd to the middle of the V-tape. Note that the time needed for this MOVEment is much smaller than the time needed for the merge (c.f. 2.5).

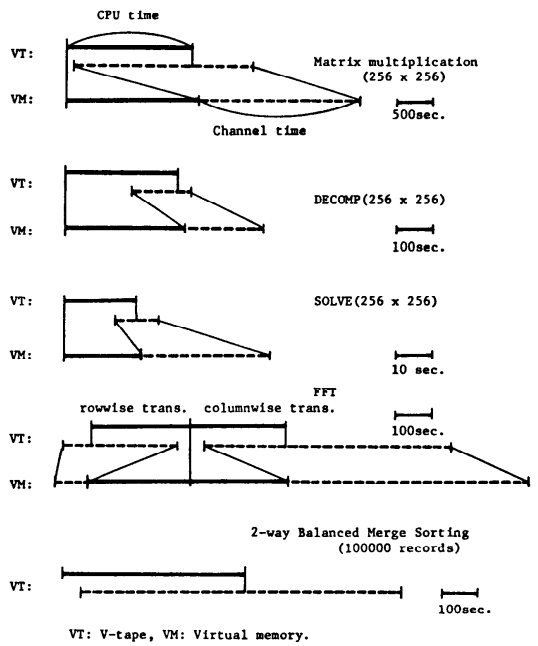

Fig. 4 Performance measurement.

Similarily, the balanced M-way merge sort can be made with 2 V-tapes and M+1 heads, while 2M tapes and 2M heads are rquired in case of magnetic tapes. Further, in V-tape systems, overhead of the "rewind" operation needed in MT (Magnetic Tape) systems is negligibly small, because we can PLACE heads to the BOT of the V-tape or MOVE heads with a reasonable speed. There is no need to devise the algorithm such as "oscillating" sort which eliminates the rewind of magnetic tape and use the "backward read/write" facility. Therefore, we can perform the sorting in higher efficiency with V-tapes than with MT.

The difference of performance between the algorithms with V-tapes and with MT is due to the fact that V-tape is backed up by the random access auxiliary memory such as discs. The V-tape sorting algorithms may be regarded as a specific sub-class of disc sorting algorithms [15, 16]. We now compare the performance of sorting with V-tape and with on-demand paged virtual memory, both memories being backed up by the random access auxiliary memory (discs). In conventional virtual memory, the simplest way to program M-way merge sort would be to declare 2M arrays, each of size S, the number items to be sorted. Then, we would need 2MS space. In V-tape, on the other hand, we can perform merge sorting with about S in size, because V-tapes can be erased by returning the space which is read by the head and is no longer necessary to keep. Thus, a considerable saving in the auxiliary memory space (discs are much cheaper than main memory, but their costs are still non-negligible). This erase function would also reduce the necessary swap number to 2/3 of that in the on-demand paged virtual memory, because blank data cells are attached to V-tape automatically, and there is no need for swap in of data for the head which performs write operation. Similar saving would be possible in conventional virtual memory by dividing the data space into small dynamic arrays, but this would result in a program almost equivalent to a software simulator of a V-tape machine.

(2) Performance Measurements

A two-way merge sort was made on the V-tape with 100,000 items, each, 16 bits long random number whose key length is 14 bits. It needs 200 K bytes of storage, which seems to be a reasonable data size to be handled by the V-tapes. Swapping time was absorbed in the CPU time as diagrammed in Fig. 4.

5. Conclusion

We built a microprogrammed processor to realize a V-tape system. We also built an on-demand paged virtual memory system using the same microprogrammed processor in order to make a comparison between the two systems for performance evaluations. As described earlier, use of the V-tapes greatly simplifies programs involving stacks, queues and I/O buffers and makes swap time sufficiently absorbed in the processing time of CPU. Matrix operations, fast Fourier transform, linear equation solving and sorting programs with much larger data area than the main memory, were shown to run more efficiently on V-tapes than in the on-demand paged virtual memory environment. V-tapes would be also applicable to the solution of partial differential equation. Further, it would be possible to perform a calculation of very long precision such as e, π to 100,000 digits efficiently with V-tapes. Especially in case of two dimensional FFT, although many hardware architectures have been developed, there seems to have been little support for auxiliary memory management. The movements of the heads during the execution of these programs are more complex than those in stacks and queues. In these cases, it is necessary to transform the algorithm into a form suitable for the V-tape system. Although the transformed algorithms are usually more complex than the original one, the elapse and the CPU idle times are reduced considerably without multiprogramming in a small system.

The V-tape scheme may be regarded as a system consisting of a tape type of data structures and operations on V-tapes which explicitly specifies prepaging requests for improving the performance of paged virtual memory systems. Therefore, the application of V-tape concept to larger multiprogrammed systems may reduce swap waiting time, the frequency of the control transfers among processes, the overhead (both in time and main memory space) of OS and user's turn around time. The multiplicity of multiprogramming may be reducible without increasing the CPU idle time, thereby improving the utility of the main storage.

No attempt has been made in this paper to propose high level language suited for handling the abstract data types corresponding to Turings tapes and heads. It would be interesting to pursue the design of such language. The design of compiler for an already existing language (e.g. FORTRAN) which automatically generates V-tape codes whenever possible would be another interesting and challenging theme.

Acknowledgement

The author would like to express his sincere thanks to Professor T. L. Kunii and Dr. Kawai for their useful suggestion, and also to Mr. T. Ida, Mr. L. A. Mateev, Mr. M. Ikawa, Mr. K. Ishihata for their helpful cooperation and discussions.

References

1. TURING, A. M. On computable numbers with an application to the Entscheidungproblem. *Proc. London Math. Soc.* (1936).
2. RICHARDS, M. BCPL: A tool for compiler writing and system programming. *SJCC* (1966) 557–566.
3. GOTO, E., ITANO, K. and MATEEV, L. A. Propositions for virtual multihead multitape processor. *Proc. of 15th Annual Conference of IPSJ*, Kyoto, Japan (Nov. 1974) 1–2.
4. SASSA, M. and GOTO, E. "V-tape", a virtual memory oriented data type, and its resource requirements. Research Report no. C-9, Tokyo Institute of Technology (Jan. 1977).
5. MATEEV, L. A. A proposal for virtual multihead and multitape processor. M. S. Thesis, Department of Physics, Univ. of Tokyo, Japan (Jan. 1975).
6. ITANO, K., IDA, T., IKAWA, M. and ISHIHATA, K. Virtual multihead multitape processor. *Proc. of 16th Annual Conference of IPSJ*, Tokyo, Japan (Nov. 1975) 97–98. (In Japanese)
7. ITANO, K. Realization of a processor with virtual tapes and its evaluation. Doctor of Science dissertation, University of Tokyo (Jan. 1977).
8. CHENEY, C. J. A non-recursive list compacting algorithm. *CACM*, **13**, 11 (Nov. 1970) 677–678.
9. KNUTH, D. E. *The Art of Computer Programming I*, Addison-Wesley (1968).
10. GOTO, E., IDA, T. and GUNJI, T. Parallel hashing algorithms, Information Processing Letters, 6 (1977) 8–13.
11. ELSHOFF, J. L. Some programming techniques for processing multi-dimensional matrices in a paging environment. *NCC* (1974) 185–193.
12. FORSYTHE, G. and MOLER, C. B. *Computer Solution of Linear Algebraic Systems*. Prentice-Hall (1967).
13. COOLEY, J. W. and TUKEY, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **19** (1965) 297–301.
14. Singleton, R. C. A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage. *IEEE Trans.* **AU-15**, 2 (June 1972) 91–98.
15. LORIN, H. *Sorting and Sort System*. Addison-Wesley (1975).
16. BRAWN, B. S., GUSTAVSON, F. G. and MANKIN, E. S. Sorting in a Paging Environment. *CACM*, **13**, 8 (Aug. 1970) 483–494.

(Received March 12, 1977; revised October 11, 1978)