A study of processing repetition‡

J. R. Haley* and A. S. Noetzel†

The phenomenon of repetition in the workload of a computer system is investigated from the point of view of its implications for systems design. Measurements taken from a CDC 6000 Series interactive system are presented. The first results show the distribution of repetitions of command executions (or program runs) over the classes of programs within the system. Then, a resource demand sequence is defined as the detailed pattern of hardware resources required by a program in execution. A technique for measuring the repetition of resource demand sequences is demonstrated, and the resulting measurements of this form of repetition are presented. The results show that the amount of repetition, though considerable, is insufficient to support the development of predictive schedulers. But they also indicate that a new approach to compilation—the use of an incremental recompiler—may effectively increase the productivity of an interactive computer system.

(Received April 1978)

1. Introduction

It is a fundamental assumption in the design of operating systems that the pattern of resource utilisation by each user program is a random phenomenon. However, reflection on the modus operandi of the users of a computing facility suggests that since each program is run many times during its period of development and productive use, the sequences of the resource demand generated during any run of a program may be quite similar to those generated during the preceding and succeeding runs of that program. An awareness of the existence of repetitiveness in a computer system workload may lead to new approaches to operating system design. For example, one might consider strategies for bypassing duplicated logical processing sequences (repeated executions of subroutines with unvarying data) in much used applications programs such as compilers. Alternatively, if the resource utilisation patterns are known to be repeated to a sufficient degree, it may be feasible to develop predictive scheduling algorithms, which schedule programs on the basis of the records of their previous runs.

In this paper, we report the results of a study undertaken to describe and measure the repetition of the patterns of demand for computer system resources. Since this aspect of program behaviour has been largely ignored, we first seek an intuitive understanding of it. A coarse yet practical quantification of the phenomenon seems a reasonable first step. We have attempted to retain sufficient generality in our approach, so that our results will be useful to evaluate or suggest design possibilities in several application areas. However, the results necessarily reflect the entire context of the study: the hardware, operating system, job characteristics, and user population. We therefore describe the relevant details of the environment in which our measurements were taken. As an example and an aid in interpreting our results, we will consider the repetition of resource demand from the point of view of the requirements of a predictive scheduler.

In the second section of the paper, we partition the commands of an interactive system into various classes, and characterise the user behaviour in terms of command usage frequencies and processing requirements over the command classes. This data is presented as a background for the study of repetition of program executions. For a more comprehensive study of user behaviour in an interactive system, we refer the reader to Boies (1971; 1974). The consistency of our results with those of Boies is apparent, even though the IBM system he has studied is quite different from ours. We then show the frequencies of the repetition of commands with unvarying file name parameters. Command repetition is expected to be the source of repetitious resource demand patterns.

In Section 3, we define the concept of a resource demand sequence, and present an example. We then discuss the technique for measuring repetition of resource demand sequences. In Section 4, we show results indicating the degree and distribution of resource demand repetition that may be expected in the workload of an interactive system. About half of all the resource requests occur in sequences that are repetitious, and therefore predictable.

In the last section, we discuss the feasibility of several techniques for exploiting the repetitiveness. It does not appear likely that predictive schedulers can be made practicable in a general purpose computing environment. But the results also show that the resource demand repetition is concentrated in the compilers and user-written programs. This strongly indicates the possibility of redesigning compilers to work in an incremental mode, thereby eliminating much duplicated processing.

2. Measurement of program repetition

As a first step in the measurement of repetition, we identify and measure the frequency of repeated use of each program by a specific user. We regard the potential usefulness of the measurements of repetition to be greatest in interactive systems, because of the great deal of involvement that such systems have with the management of the user programs and files. Within interactive systems, there exists the possibility of greater integration of the schedulers and systems software that represent a major portion of the processing activity. Our study was conducted on the CDC 6400 with the UT-2D operating system at the University of Texas. The workload of this system consists of programs initiated by users at interactive terminals. This interactive system was developed from a batch operating system. It has an advantageous structural simplicity, which allows the collection of data describing user behaviour at several different levels, and enables the identification of various job types or processing modes within the system. We will distinguish these processing modes and report our results for each category. We first present an overview of the interactive system.

‡The research reported in this paper was sponsored by National Science Foundation grant GJ-39658.
*Department of Computer Sciences, The University of Texas, Austin, Texas 78712, USA.
†Applied Mathematics Department, Brookhaven National Laboratory, Upton, New York 11973, USA.

The interactive system

The UT-2D interactive system evolved from an elementary facility that simply allowed control cards to be entered from an interactive terminal. The command vocabulary available to the user has now been expanded to include many functions specifically for interactive use, but a relatively small subset of the basic control card commands still represents the major portion of computation in the interactive system.

The file system resides on a hierarchy of storage devices. The storage medium at the lowest level is magnetic tape. Each tape holds a set of files, called a permanent file set, belonging to a single user. When a file in a permanent file set is referenced, as by the interactive command READPF, the entire permanent file set is transferred from the tape to the next level of the hierarchy (if it is not already present there), which is a set of auxiliary disc units. The requested file is read from the auxiliary disc unit to the highest level, the set of four large system disc units, and it is then logically attached to the user job as a local file. Any program requested by the user may reference the local file just as it does the standard local files INPUT and OUTPUT. Once modified, a local file may be restored to the permanent file set by the command SAVEPF. Various forms of the COPY command write the contents of one file to another.

The interactive system is structured as a command interpreter. Commands entered at the terminals may name system routines, or they may name binary files attached to the user job as local files. Commands, in this system, are synonymous with programs. For a profile of the usage of the interactive system, it is convenient to partition the commands available to users into the following classes:

1. *Utility* commands. Many of the commands in this class, such as READPF, SAVEPF, COPY, etc. specify various file positioning and data transfer operations. Others control file names and job parameters.
2. *Editors* and interactive compilers. There are several text editors in the system: the one most often used is called EDIT. The BASIC compiler has an interactive mode which can be distinguished at the command level. These software processors are characterised by a large amount of interaction with the terminal once the command has been entered.
3. *Compilers* and other applications software. FORTRAN and BASIC are the most heavily used languages in this system. The BASIC compiler is included in this class when it is not in the interactive mode.
4. *User programs.* Used as a command, a file name causes the loading and execution of the program within the file. Almost all of the commands not in the above three classes call for the execution of user-written binary program files.

Distribution of processing by command type

Table 1 shows the frequency of use and average processing time requirements of the command classes. The processing time requirements are shown as two components: the CPU time and the peripheral processor time, in seconds. Most of the peripheral processor time represents data transfers for I/O operations. The Utility commands are seen to be I/O bound, whereas the other command classes have an approximate balance between CPU and I/O requirements.

Table 1 Processing requirements for command classes

Command class	*Command frequency*	*CPU processing per command*	*Peripheral processing per command*	*Percent of system processing*
Utility	Ø.744	Ø.ØØ39	1.12	18.4
Editor	Ø.112	Ø.7Ø3	6.45	2Ø.6
Compiler	Ø.ØØ8Ø	4.31Ø	1.87	39.2
User program	Ø.Ø64	3.15Ø	1.Ø1	21.8

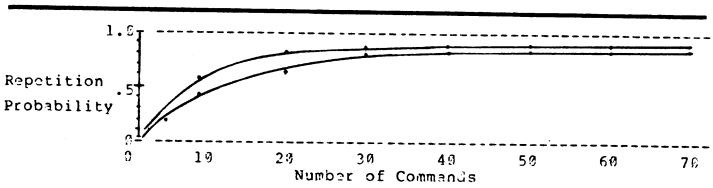

Fig. 1 Probability of detailed repetition of the EDIT and FORTRAN commands

Table 1 also shows the percentage of the total system processing (CPU time plus one sixth of the peripheral processor time) accounted for by each of the command classes. Though the Utility commands have by far the highest frequency of use, their processing requirements are so small that the amount of processing within this class is comparable to that of the other classes.

Detailed repetition of commands

The system accounting file contains a record of the commands and the parameters of the commands issued by the interactive users. Through searching the accounting file, instances of potentially repetitive program executions were found. A *detailed repetition* of a command is said to occur each time an interactive user repeats a command with identical input parameter file names. Parameters that are not file names are not considered in detailed repetition. In searching for detailed repetition, adjustments are made for several common file names that often signify different files.

The probability that a command is repeated in the detailed sense increases with the total number of commands issued by the user. **Fig. 1** is the cumulative probability distribution for the number of commands issued between detailed command repetitions. The characteristics shown are for the EDIT and FORTRAN commands. The remainder of the interactive commands have detailed repetition characteristics similar to those shown in Fig. 1. Generally, the commands with the highest frequency of use also have the highest probabilities of detailed repetition.

The data of Fig. 1 was accumulated from the occurrences of these commands for which at least 70 of the user's preceding commands (possibly over several interactive sessions) appeared in the accounting log. We define the *command repetition factor* (CRF) of each command type to be the final observed value of the probability of detailed repetition. As the characteristics of Fig. 1 show, the final value for the observed data is approximately the limiting (in the mathematical sense) value of the detailed repetition probability. The difference between one and CRF is the probability that the command is not a repetition. The CRF values for various commands are shown in **Table 2**, column 2. These values represent an upper bound on the degree of repetitiveness of the processing associated with the various commands.

Resource demand sequences

A *resource demand sequence* of a program execution is a detailed profile of the hardware resources required by the program during its execution. Ideally, this profile should be a consequence of only the program and its data: it should be free from the effects of the scheduler and the multiprogramming mix in which the program is run.

One way to obtain this characterisation of a program execution is to run the program in a uniprogramming mode and record as the resource demand sequence the times of the CPU processing periods together with the memory, I/O and other

Table 2 Repetitiveness factors

Command k	n_k	CRF_k (%)	SRF_k (%)	V_k (%)
READPF	70	83	85	21.3
REWIND	44	92	97	2.6
RETURN	41	42	100	1.9
SAVEPF	42	90	78	11.6
SHOW	27	67	63	1.6
49 Others	223	53	63	14.9
Total: UTILITY	447			53.9
EDIT	129	92	5.2	4.3
BASIC	2	90	3.1	0.1
TEXEDIT	2	84	3.8	0.2
Total: EDITOR	133			4.6
FORTRAN	67	87	84	31.3
SPSS	41	67	71	1.6
BASIC	33	84	75	7.0
PASCAL	9	82	88	7.6
COBOL	17	74	81	6.2
24 Others	33	69	80	11.6
Total: COMPILER	200			65.3
USER PROGRAM	107	91	80	80.1

supervisor service requests as they occur. But even with this straightforward approach, it can be seen that the definition of resource demand cannot always be fully separated from the modes of the system's response to that demand. In a virtual memory system, for example, the series of page faults indicating the memory demand will be determined by the system's page replacement policy, and that policy may well depend upon the activities of other programs.

But it is not necessary to belabour this point. Since our intent is only to obtain estimates of the degree of repetitiveness of sequences (at the level at which the hardware resources are scheduled), any resolution of these ambiguities that can be applied consistently will be useful.

Our technique for extracting resource demand sequences involves tracing programs in their true multiprogramming environment. We will supplement our informal definition of the resource demand sequence with an example drawn from the CDC 6000 series machine used in this study. We expect that the examples and data drawn from this system will convey the essence of our motivation, technique, and results.

The user programs in the CDC 6400 system are multiprogrammed in a main memory of 64000 60-bit words, and executed by a single fast CPU. A pool of seven peripheral processing units (PPU's), which are twelve-bit minicomputers, each having a memory unit of 4000 twelve-bit words, execute auxiliary functions, including I/O, for the user programs. The PPU's are ideally suited for control functions, while the CPU is a slave. However, some service functions reside in a portion of the operating system called CMR (central memory resident) and are executed by the CPU. One PPU, called MTR (monitor), constantly polls both the user program active on the CPU and the active pool PPUs, to determine whether operating system service is needed. Upon finding a service request, MTR either executes the required function, assigns it to an available PPU, or, if it is a CMR function, interrupts the CPU. MTR also interrupts the CPU at the end of each processing quantum to effect a round-robin schedule of service to user programs. Pool PPUs that require CMR service are also capable of interrupting the CPU.

Thus, the system may be viewed as consisting of four virtual processors: the user programs, MTR, the set of pool PPUs, and the CMR portion of the operating system. Each request for system resources is made via communication between these processors. The system has been equipped with a detailed trace facility that can record all interprocessor communication events. While the trace is in operation, each communication event is marked with timing data and written in a main memory buffer. The buffer is periodically dumped to a magnetic tape, called the *system event trace*. Under moderate load, a complete tape is obtained in about half an hour.

The user program resource demand sequences are obtained through offline processing of the system event race by a program called the *trace decomposition program*. In addition to accumulating the events corresponding to each job, this program filters out all of the effects of the scheduler and the multiprogramming mix. It adjusts the times of each jobs' events to be relative to the processing time of the job, eliminating the queueing periods and other delays due to the system. It removes all events initiated by the operating system, such as memory swap-in and swap-out sequences for pre-emptive memory scheduling, and it links the various processor runs to the events that invoked them.

An example of a resource demand sequence

A simplified example of a resource demand sequence is shown in **Fig. 2.** The elements of the sequence are (time, event) pairs. In this example, the FORTRAN compiler (RUN) is executed. It calls for an I/O operation after three milliseconds of processing, and then halts for I/O completion after fifteen milliseconds. The PPU sequence corresponding to the I/O call requests and uses device three (REQ) of channel two (RCH), and then, after 86 milliseconds, recalls the central program. The central program requests a 50,000 word block of memory in the RFL (request field length) call after 30 milliseconds of its processing.

Stability of the recording technique

The representations of user program resource demand sequences produced by the trace decomposition program have been shown to be stable: the CPU sequences obtained by identical runs of a program differ only by minor variations of the timing data. The timing variations are a result of deficiencies in our measurement tools. For example, the precision of the clock used in the trace facility was limited to a quarter millisecond, whereas the average period of uninterrupted CPU processing was in the order of two milliseconds. We also note that many events are recorded only after they are recognised by MTR, hence after a variable delay of up to two milliseconds for the MTR polling cycle. Memory interference due to simultaneous I/O operations with processing also has a noticeable effect on the apparent length of each computation.

The PPU sequences show the channel and device holding times for I/O activity. This timing data was more variable than

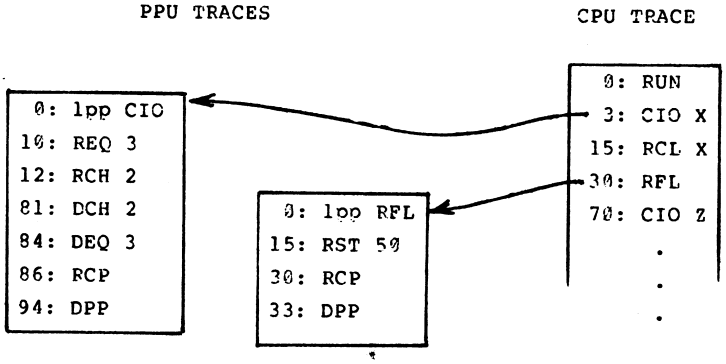

Fig. 2 Example of resource demand sequence

that of the CPU sequences. Since the delay for disc arm positioning and latency was not recorded separately from data transfer time, these delays could not be filtered out of the resource demand sequences. The only other significant effect of the operating system that could not be eliminated is the apparently arbitrary assignment of the user program files to the various I/O devices. The device numbers specified for I/O operations therefore did vary under identical executions of a program.

Repetition of resource demand sequences

In considering a resource demand sequence to be a repetition of another, we allow for little variation in the sequence beyond that due to the imprecision of our measurement tools. A resource demand sequence is defined to be a *repetition* if it has been preceded by another, its *predecessor*, such that the following conditions hold:

1. Each event in the CPU processing sequence matches, in kind, the corresponding event in the predecessor sequence.
2. Each CPU processing period is within fifteen percent of the length of the corresponding period of the predecessor sequence.
3. All events of the PPU sequence match, in kind.

The search for repetition of resource demand sequences is limited to detailed command repetitions. When a detailed command repetition is found during a period in which the trace was in operation, the resource demand sequences of both the repetitive use and that of its predecessor were extracted. When multiple detailed repetitions occur, those between the first and the last play the role of predecessor as well as repetition.

For the i^{th} detailed repetition of command type k (READPF, EDIT, etc.), let P_{ki} and R_{ki} represent the accumulated CPU time in the resource demand sequences of the predecessor and the repetition, respectively.

The comparison program performs a tree search, scanning the CPU sequence, and each PPU sequence in the order in which it is invoked in the CPU sequence. It compares events of the predecessor and repetition resource demand sequences and halts when it finds a violation of the above rules. Let C_{ki} be the accumulated CPU time at which an irreconcilable difference between the traces occurs: if there is none, then $C_{ki} = R_{ki}$. The *sequence repetition factor* (SRF) for each command type is defined as the ratio of the total repetitive processing time to the total processing time in the repetitive uses of the command:

$$SRF_k = \frac{\sum_{i=1}^{n_k} C_{ki}}{\sum_{i=1}^{n_k} R_{ki}}, \qquad (1)$$

where n_k represents the number of detailed repetitions found in all of the periods in which the system trace was active. Because the processing times are summed over all repetitive uses of the command, the repetitiveness of each program run is weighted by the length of the run in the computation of the SRF. The SRF shows the fraction of processing that may be predictable within all the repetitive uses of the command.

4. Results: resource demand repetition

Resource demand sequences of all the interactive jobs in the system were collected, by the means outlined in the previous sections, from five trace periods, taken on different days at various times. In all, 57 complete and 107 partial interactive sessions representing 133 different users were recorded.

The SRF values for several of the system commands are given in Table 2, column 3. Most notable is the great disparity between the SRF value of the Editor class and that of the other three classes. For all commands except the editors, the input data is in the files named as parameters of the commands, with a negligible amount being supplied by interaction with the terminal. But the program traces show that each EDIT call is followed by about ten interactions with the terminal. The interactiveness with the editors is of the kind that calls for various processing functions (as opposed to numerical data), hence, results in divergent processing sequences at the various invocations of the editor.

The product of CRF_k and SRF_k represents the fraction of processing due to command type k that is repetitive. We obtain repetitiveness figures for each command class and the system as a whole as follows. Let F_k represent the ratio of the processing that occurs within command type k to the total processing of its class. Then

$$V_k = CRF_k * SRF_k * F_k \qquad (2)$$

is the contribution of command k to the total repetitiveness of its class. By summing V_k over all the commands of the class, one obtains the fraction of all the processing within the class that is repetitive. These results are shown in Table 2, column 4.

The Utility commands show moderate degrees of repetitiveness, although the values were broadly distributed over the various commands. The most significant result is the high repetitiveness factors of the Compiler and User Program classes. Weighting the repetitiveness factors of each class by the fractions of system processing within the class (Table 1, column 4) and summing over the classes, one obtains the fraction of all system processing that is repetitive. This result is 52 percent.

5. Conclusions

The goals of this study were to define appropriately and then measure repetitiveness of resource demand sequences, and to examine its implications for systems design. One possibility that was considered is that the repetitiveness present in the workload of an interactive system might be a fruitful source of predictive information to be used in scheduling. Our definition of resource demand repetition is consistent with the requirements of a scheduler that would predict the future resource demand pattern of a program to be the same as that of a previous run. If the repetitiveness were 100 percent, the pattern of requests by the previous run would be a perfect predictor. In that case, a deterministic scheduler could be designed to achieve the optimum throughput.

Since perfect predictability is not possible, we undertook simulation studies of various straightforward predictive CPU and memory scheduling algorithms, with varying degrees of correctness of the predictive data. Examples of such algorithms are described as 'shortest CPU burst time first', 'shortest time to completion first', 'shortest time to I/O operation first', etc. The results showed that properly designed predictive algorithms could achieve higher device utilisations and throughputs than standard algorithms (Noetzel, 1974). The margins of difference, however, were only a few percent, because under proper loading, the simulated system is capable of high CPU utilisation (Sherman, Browne and Baskett, 1971). But more significantly, the predictive algorithms required a high degree of correctness of the predictive data in order to maintain their superiority. When the correctness (that is, the CRF*SRF product) was degraded to about 70 percent, the predictive algorithms lost their advantage over standard algorithms.

Although better techniques for recording and exploiting repetition may be developed, the overhead of the scheme, which is likely to be considerable, must also be taken into account. On balance, we must conclude that predictive scheduling in this environment is not advantageous. Noting the distribution of repetitiveness over the command classes, we may speculate that in a batch processing environment in which a small number of large programs are run repeatedly, some variation of the

predictive scheduling technique may be beneficial. In Forbes and Goldsworthy (1977), for example, the quantifications of CPU boundedness and I/O boundedness are expected to be of sufficient importance to determine an efficient scheduling mix. Our results indicate than an even more specific representation of dynamic resource requirements is available.

One positive result for computer systems design can be drawn from this study. The high degree of resource demand repetitiveness in the Compiler class suggests that the compilers are performing a large amount of logically repetitious processing; that is, executing subroutines or processing phases with unvarying data. It seems apparent that a significant degree of processing could be eliminated by redesigning compilers to operate in an incremental mode, recompiling modified statements only, and integrating these with the stored object code of the previous compilation. In an interactive system, the changes are made through the text editor: they could be made known to the compiler with no explicit directives by the programmer. Using the data for resource demand repetition presented here, eliminating only half of the repetitive processing of the FORTRAN compiler reduces the total processing workload of the system by over six percent.

The high degree of repetitiveness in the User Program class indicates a potential benefit in studying their logical repetition also. Techniques for automatically eliminating a large amount of duplicated processing may become apparent after the most frequent repetition patterns are characterised.

References

BOIES, S. J. and GOULD, J. D. (1971). User Performance in an Interactive Computer System, *Proceedings, Fifth Princeton Conference on Information Sciences and Systems.*

BOIES, S. J. (1974). User Behavior in an Interactive Computer System, *IBM Systems Journal*, Vol. 13 No. 1.

ESTRIN, G., MUNTZ, R. R. and UZGALIS, R. C. (1972). Modelling, Measurement and Computer Power, *Proceedings, Spring Joint Computer Conference.*

FORBES, K. and GOLDSWORTHY, A. W. (1977). A prescheduling algorithm—scheduling a suitable mix prior to processing, *The Computer Journal*, Vol. 20 No. 1.

HOWARD, J. (1973). A Large-Scale Dual Operating System, *Proceedings, ACM National Conference.*

JOHNSON, D. S. (1972). *A Process-Oriented Model of Resource Demands in Large Multiprocessing Computer Utilities*, Ph.D. Dissertation, The University of Texas.

NOETZEL, A. S. (1974). Simulation Studies of Predictive Scheduling, Technical Report No. 37, Department of Computer Sciences, The University of Texas.

NOETZEL, A. S. and HERRING, L. A. (1976). Experience with Trace-Driven Modeling, *Proceedings of the Symposium on the Simulation of Computer Systems.* National Bureau of Standards, Boulder, Colorado.

NOETZEL, A. S. and HALEY, J. R. (1975). *The Decomposition of a Multiprocessor Event Trace into User Program Resource Demand Patterns*, Technical Report No. 49, Department of Computer Sciences, The University of Texas.

SHERMAN, S. W. (1972). *Trace Driven Modelling Studies of the Performance of Computer Systems*, Ph.D. Dissertation, The University of Texas.

SHERMAN, S. W., BROWNE, J. C. and BASKETT, F. (1971). Trace Driven Modelling and Analysis of CPU Scheduling in a Multiprogramming System, *Proceedings, SIGOPS Workshop on System Performance and Evaluation*, Harvard University.

Book reviews

Advances in Information Systems Science (Volume 7), edited by Julius T. Tou; 1978; 310 pages. (*Plenum*, £17·32)

This is the latest in a series of books edited by Tou which have appeared roughly annually since 1969. The book consists of five articles apparently solicited by the editor. Although each chapter contains a background introduction, the papers are by no means tutorial such as those in *Computing Surveys*. There is no obvious theme addressed by all the papers and the majority of readers will find at most a couple near to their interest. The work presented is of uneven quality, and the whole is not up to journal standards. Presumably the book is intended for libraries.

The first paper surveys fault diagnosis techniques for computer hardware and discusses some practical and useful diagnostic methods in real world digital systems. The second chapter presents a design for a heterogeneous, widely distributed, heavily used information network that permits complete sharing using current systems. That is quite a tall order! The proposal consists of a set of Basic Modular Units (BMUs) communicating via a single switching system. The BMUs consist of a large central configuration surrounded by satellite minicomputer systems. The system implementation language is a variant of FORTRAN. The third paper is an interesting attempt to develop a mathematical model of a distributed information system on which to implement a distributed data base. The problems are framed in terms of constrained optimisation for which heuristic approaches are proposed. The fourth paper discusses the revolution in computing which has been caused by the recent dramatic shift in hardware costs. The author argues that a computer should be large enough to support a limited application with just a small number of users. Additional users should be catered for by multiple copies of a system rather than additional system complexity, an approach which should only be used in conjunction with a narrowing of applications supported. The final paper muses at length on the role of data structures in pattern recognition.

J. M. BRADY (Colchester)

Advanced Programming Techniques by C. E. Hughes, C. P. Pfleeger and L. L. Rose, 1978; 287 pages. (*John Wiley*, £10·50)

A large balloon blown up in a small box will have its inherent shape modified. If this book is less well rounded than it might be, certain constraints might be supposed. Perhaps the needs of the authors' students have been one such constraint, changing what could have been a book on advanced FORTRAN programming into one which also includes information on JCL and internal data representations for IBM and DEC machines.

Perhaps the authors' limited experience with few types of computer imposes the greatest constraint on the book. Despite lipservice paid to them, the FORTRAN standards receive scant attention. Somewhat naive statements are made at times ('A DO-loop always executes at least once' p. 27).

These are unfortunate limitations on a book which contains much useful material on programming techniques. The advice given concerning program design and documentation is excellent. The subject areas (such as data structures) are demonstrated with clear sample programs and well chosen exercises. If your constraints resemble those of the authors, you may well find this book very valuable.

A. C. DAY (London)