Abstract:

This report has been prepared at the request of the Science Research Council. It discusses the needs for, and the problems of achieving, high reliability from complex computing systems, and includes a brief survey of several such systems.

Highly Reliable Computing Systems

by

B. Randell
RANDELL, Brian.

Highly reliable computer systems. By B. Randell.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1971.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 20)

12"

Added entries


Suggested classmarks (primary classmark underlined)

Library of Congress:
Dewey (17th):
U.D.C.: 621.381952 332.1 387.74
681.322.06 : 332.17 : 351.814

Suggested keywords

HARDWARE FAULTS
ON-LINE SYSTEMS
PROGRAM ERRORS
PROTECTION
SYSTEM RELIABILITY

Abstract

About the Author

Professor Randell has been Professor of Computing Science in the Computing Laboratory of the University of Newcastle upon Tyne since 1969, before which he was at the I.B.M. Watson Research Centre, U.S.A.
## CONTENTS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>System Performance and System Reliability</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Hardware Reliability</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Software Reliability</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Coping with Unreliable Components</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Generic Software</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Acknowledgements</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>References</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Appendix 1 – The BOAC On-line Airline Reservations System</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Appendix 2 – Barclays Branch Accounting System</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Appendix 3 – Barclays On-Line Banking System</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Appendix 4 – The STL Dependable Process Computer</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Appendix 5 – The BCC-1 System</td>
<td>37</td>
</tr>
</tbody>
</table>
1. Introduction

This report has been prepared at the request of the Computing Science Committee of the Science Research Council. The intent of the report is to discuss the needs for, and the problems of achieving, high reliability from complex computing systems.

For the purposes of this report is is quite adequate to take the term 'system reliability' as meaning the trustworthiness of the results produced by a system. What is often called 'system availability', i.e. the extent to which continuity of system operation is achieved, can be regarded as being an aspect of system reliability. However, it can also be regarded as being involved with system performance. In fact, a discussion of the interactions between system performance and system reliability is given in section 2 of this report.

During the preparation of this report, a brief survey was undertaken of several large computing systems in Britain which have an obvious requirement for very high reliability. These systems, namely the BOADICEA on-line airline reservations system of BOAC, and the branch accounting and on-line banking systems of Barclays Bank, are discussed in Appendices 1, 2 and 3 respectively.

Two other environments that we originally thought to include in our survey were air traffic control and on-line hospital patient monitoring. However, on somewhat closer investigation it appeared that, although both these areas may in the future have extremely critical needs for computing system reliability, since the lives of human beings could be at stake, this time is as yet some way off, at least in Britain. For example, we gather that the medical aspects of on-line patient monitoring are still very much a subject for research, and the most that would be expected from a computer for several, perhaps many, years to come would be for it merely to supplement conventional nursing techniques. However, one should remember that situations, and opinions, sometimes change rapidly.

In this report attention is, in the main, concentrated on large on-line real-time systems. This is not meant to imply that these are the only environments which have critical requirements as regards the trustworthiness of the results produced by a computing system. To quote Graham [17], whom I know to have been deliberately understating the case, 'It is my understanding that an uncritical belief in the validity of computer-produced results (from a batch-processing computer) was at least a contributory cause of a faulty aircraft design that led to several serious
air crashes'. Similarly, it is not necessary for a system to involve on-line facilities, or to be what one would normally think of as a 'real time' system, for there to be requirements for high availability, as the discussion of the Barclays Branch Accounting System, given in Appendix 2, makes clear. However, the difficulties of obtaining high reliability from complex on-line real-time systems are often immense, and it is no light matter to allow oneself to reach the situation of requiring extreme reliability from such a system. Thus, the paper by Rodberg [29] describing the reliability requirements of the computing system involved in SAFEGUARD, the U.S. government's planned anti-ballistic missile system, makes frightening reading, particularly when one also reads the companion paper by Licklider [24], entitled 'Underestimates and Overexpectations', which surveys several earlier complex computing systems.

The present report consists of several sections which attempt to provide an overview of some of the problems of obtaining high reliability from complex computing systems, together with appendices which describe several systems of interest in this regard. (The material in some of the early sections is an expanded version of part of a survey paper [28] prepared by the author for IFIP Congress 71.) Sections 3 and 4 discuss the problems of hardware and software reliability, respectively, and section 5 concerns the problems of designing a system that can continue to provide reliable service in the presence of hardware failures and software errors. The rather special problems that are caused by system software that is designed not for a single installation, but rather for a whole range of environments and machine configurations, are discussed in section 6.

2. System Performance and System Reliability

System performance and reliability are both 'commodities' which are of value to users, and whose 'production' will involve the incurrence of costs. Enough different computing systems have been produced and installed that one can attempt to quantify the relationship that holds between system performance (however crudely this might be measured) and cost — Grosch's 'law', that performance is proportional to the square of cost, is a well known example. Such a relationship may tell us more about a certain manufacturer's pricing policies than the realities of his development and manufacturing costs. However, it does indicate that there is at least a certain level of understanding amongst customers of the need to assess their performance requirements, and of how the performance that they
obtain from a system is, or should be, related to the amount that they paid for it. The situation with regard to system reliability is very different. A really naive user will not realise just how unreliable both the hardware and the software of the computing system that the manufacturer delivers to him might be. Most users will be hard put to quantify the value that they place on obtaining a certain level of reliability, let alone have any idea how best to allocate the money that they wish to spend in order to obtain this level of reliability.

The specific role of the operating system in all this is rather interesting. Ideally, the task of the operating system is that of enabling an installation to achieve the inherent performance capabilities, and surpass the inherent reliability capabilities, of the basic hardware. Of course it is not unknown for the amount of resources used by the operating system itself (CPU time, storage, etc.) to be so great as to cause one to question whether it is in fact making a positive contribution to the capacity of the computing system to produce useful work. Similarly, an operating system may contain so many errors that these errors become the dominant factor in the overall reliability of the computing system, rather than hardware failures, whether or not these failures are dealt with adequately by the operating system.

There are, needless to say, interactions between performance and reliability. Reliability is at least in part bought at the expense of performance — precautionary measures such as attempts to detect the occurrence of errors, multiple recording of data, etc., all use up resources and can impact performance. Conversely, the lack of reliability at some point within a system, can sometimes be dealt with by the system, using fall-back and recovery techniques, so that the problem manifests itself to a user of the system simply as reduced performance. It is when an error goes undetected that the system will produce untrustworthy results, or perhaps no results at all (which can of course also happen even if the error is detected, if the system is not capable of coping with the situation).

Clearly, operating system designers have to be aware of these interactions between performance and reliability, and must attempt rational trade-off decisions. (A result that is 'guaranteed' correct, produced after all need for it has passed, may well be less valuable than a timely result which has some (hopefully small, and known) probability of being incorrect!) These decisions are very difficult for two reasons. Firstly, in our present state of knowledge, it is often difficult to predict what
impact a particular feature, which is intended, say, to improve computing system reliability, will have on either reliability or performance. Secondly, it is very difficult, even when these impacts can be predicted, to judge whether the feature is worthwhile, in view of the lack of accepted norms for relating reliability to cost or to performance. Of course, the reality is that both the performance and reliability achieved by the early versions of most operating systems are far from adequate, and many iterations are usually needed before adequate levels are obtained.

Two very good illustrations of this are the very successful Project Apollo Ground Support [2] and BOADICEA systems. Both systems can be seen to be absolutely direct descendants, through several predecessor systems, of systems whose first implementation was attempted many years ago. Both systems, and all their respective predecessors, went through many iterations, though not only for the purpose of improving performance and reliability, but also to cope with extended functional specifications.

Incidentally, the Project Apollo Ground Support System is also notable for the efforts that have been made to minimise the extent to which reliance is placed on the correct and continuous functioning of the system. Outputs from the system are monitored by many engineers and programmers, sitting at display screens, who have the task of deciding whether the outputs are to be relied upon — if not, they can cancel the relevant part of the system, or replace it by an earlier version. Although a vast amount of the system is programmed afresh for each mission, only a small well-tried inner core of the system is vital to the safety of the astronauts. Furthermore, brief interruptions of processing, such as are involved in switching (manually) to standby hardware systems, are acceptable. These facts in no way detract from the impressiveness of the system, which must be one of the most successful large systems in existence. However, it is significant that no attempt is made to rely solely on hardware and software, but rather that the human element still plays a vital role.

3. Hardware Reliability

Much progress has been made in achieving ultra-high reliability from hardware modules which are essentially electronic such as processors and memories. Much of this increased reliability comes from improved technologies. However, hardware failure detection and correction facilities have also developed greatly. The STL 'Dependable Computer' which
is discussed in Appendix 4 is a striking example of what can be achieved by the careful use of such facilities. In essence the computer can be described as completely reliable. Any single failure, even on the checking circuits, can be detected, the offending circuit board identified, and replaced, all without interrupting the system — the time to replace a board is infinitesimal compared to the mean time between failures.

The quest for ultra-high reliability has on recent years usually been confined to computers for military or aerospace use. Rightly or wrongly, manufacturers have assumed that the extra development and manufacturing expense involved in providing the sometimes quite high degree of hardware redundancy that is involved was not appropriate for the commercial market. The model 155 of the newly-announced IBM S/370 series is, I believe, a recent important exception to this statement. I gather that it contains of the order of 30% extra circuitry in order to provide, for example, extensive checks on the working of the arithmetic unit, and what is called an 'instruction retry' capability. This capability allows the execution of virtually any instruction to be restarted, should an error be detected, and is intended to cope with transient hardware faults. Certainly, it has been reported that the reliability of the system is strikingly improved over that of its predecessors.

Clearly, one has reason to hope that hardware failures, at least of the electronic variety, will in the future be a less important source of unreliability in a complex computing system, than it is at present in, for example, the BOADICEA system. However, it would be unwise to assume that designers of operating systems for computing systems with critical reliability requirements will in the near future be able to ignore the problems of electronic failures. It is by no means clear that the most cost-effective solution will be for all such failures to be dealt with completely by hardware error detection and correction mechanisms. Indeed, as electronic technology develops, our concept of what is the most appropriate replaceable unit is changing rapidly. On the STL computer, the replaceable unit is a circuit board which contains only a comparatively few circuits. However, large scale integration is progressing to the point where it is claimed that a single pluggable board could contain say, an entire PDP-8 processor, including memory! If the hardware was made redundant
at this level, the temptation to use the 'spare' processing power to increase system performance would be difficult to resist. It is for this reason that systems such as the ESS system [11] which involve multiple processors, performing identical actions, the consistency of which is continuously checked, are I believe not fully representative of major future trends.

The situation is worse with electro-mechanical devices, where the levels and costs of redundancy needed to achieve comparable reliability are much higher. Thus, the 'hidden' ninth spindle on an IBM 2314 disk drive, kept as a spare, although of value in increasing the probability of there being eight spindles in working order, does not prevent loss of data caused by a head crash. To do this would require duplication of the entire set of eight spindles, and that all data be automatically recorded in duplicate. It is unlikely that this would be regarded as the most appropriate way of utilising the eight extra spindles, compared to audit trail and dumping facilities such as those described by Fraser [15].

In summary therefore, there seems little chance that the operating system designer will be able to avoid taking at least some of the responsibility for coping with the consequences of either electronic or electro-mechanical failures. (Indeed, similar comments can even be made about power supply failures). However, there is a big distinction between taking responsibility for coping with an already detected hardware failure, and having to actually detect the failure. For example, software for detecting that a disk is not storing information correctly is often included in an operating system. However, one would hardly expect an operating system to be written in such a way that it could detect any failure in the instruction decoding or storage addressing hardware of the processor on which it was being executed. Instead it seems reasonable to assume that improved hardware facilities, at least for detecting, if not correcting or coping with, failed hardware modules, and for reporting and identifying these modules, so that the software can try to deal with the problem, will become more common.

One of the first computing systems which was specifically designed with the intention of being able to cope with failures in hardware processors, memories, I/O devices, etc., was the Burroughs D825 [1], developed in 1962, and used in an aircraft and missile surveillance system which was a successor to the SAGE system. One short paper [30] was produced on the AOSP, the operating system designed for this application, but military and commercial secrecy prevented publication of details of the techniques for
detecting and diagnosing hardware failures, or of re-configuring the system under program control. Since then the IBM 9020 system, which has close similarities to the D825 system, has been developed for another specialised application [19] — the F.A.A. air traffic control system. Just one brief paper [23] has been published on the Operational Error Analysis Program which is used in this system to try and cope with the occurrence of hardware failures. In addition, a paper [6], on the RCA 215 multiprocessor system, given at the session entitled 'Architectures for Long Term Reliability' at the 1969 Fall Joint Computer Conference, includes a brief summary of the error recovery techniques that were to be used. (Other papers at this session [3,4,13] contain useful background material either or the survey type, or more specifically related to the reliability of hardware components. As is so frequently the case, the problem of detecting and repairing damage to the software system that is caused by hardware failures, is largely ignored.)

As yet, systems such as the D825 and the IBM 9020, which provide what is sometimes called 'graceful degradation' or 'fail-soft' capabilities, are uncommon, and the problems of providing such capabilities are neither widely nor well understood. The value of such systems is not confined to situations where there is a requirement for continuous availability of processing capability. Rather, the ability to reconfigure the system dynamically, into many differing sets of partitions, allows many hardware maintenance operations which would otherwise require a dedicated system, to be performed on a suitable partition of the system whilst the rest of the system is used to provide service. The importance of increasing our understanding of the problems of designing such fail-soft systems, preferably in general, rather than solely in relation to specific computer designs and environments, therefore seems quite clear.

4. Software Reliability

A common view of software reliability is that it is achieved solely by ensuring that the software is correct, i.e. is free of bugs. Software bugs are seen as the equivalent of design errors in hardware, with there being no equivalent to the failures, such as are caused by component ageing, that can occur after all the design errors have been removed. This view is somewhat simplistic — for a start, the distinction between hardware design errors and later hardware failures can be somewhat arbitrary. However, what is more to the point is that it is rarely possible to wait until all the bugs have been removed from a
complex software system, before it is used to provide service. (For that matter, it is not uncommon for blatant hardware design errors to be found many years after installation of a complex computing system). Indeed, there are many who would deny the possibility of a large software system ever reaching a bug-free state. Certainly the current statistical evidence is on their side — it was recently stated that each release of OS/360, which is (one hopes) an extreme case, has on average over one thousand distinct errors reported in it.

It is worth examining what we mean by the term 'correctness'. Needless to say, the results produced by a system can only be 'correct' with respect to some criterion. One would like to assume that such a criterion would be part of the detailed specification that was used to guide the design and implementation of a system. However, for other than very simple systems, such specifications are unlikely to be accurate or complete. Rather, they are often little more than an initial bargaining offer, subject to re-negotiation as the system implementation proceeds and the designers and their customers start getting detailed feedback. Naturally, to a user, the fact that a system is correct with respect to some inadequate or obsolete specification will be irrelevant — to him it will be, in essence, incorrect.

It is against this background that the current research on the topic of program correctness should be assessed. Much of this work derives from that of Floyd [14], who proposed the use of automated theorem proving techniques to check the consistency of a program with programmer-supplied formal assertions about the relationships which should hold amongst the values of the variables at various stages during the execution of a program. This has in fact been done by King [21], but King's work, impressive though it is, makes it clear that, at the present state of development, even quite simple programs can tax the abilities of automated theorem proving techniques.

However, this technique of program verification does avoid the objection that can be made against manual efforts to provide formal proofs of the correctness of programs. The value of a proof depends on its credibility, which in turn depends on its clarity and conciseness. For example, a fifty page proof of a one page program would not be very satisfactory, to say the least. However, assuming that one was willing to have faith in the correctness of the theorem proving techniques used in King's program verifier (and in the reliability of the computer on
which it is run), then the reluctance to accept massively intricate proofs might be overcome.

The work by Naur [25, 26] and Dijkstra [8, 10] on program verification has been described as using a 'constructive approach'. The provision of assertions, and the checking of their consistency, is regarded as part of the task of programming, and as being of value to a programmer in guiding him towards the production of a correct program. The work of Dijkstra and his colleagues on the THE system [9], who took as their goal the task of satisfying themselves, a priori, as to the 'correctness' of their design for a multiprogramming system, is of course well known. However, the technique of 'structured programming', which Dijkstra advocates as a discipline for programmers to use, is by no means a panacea, as has been shown by Henderson and Snowdon [19]. For all this, the structuring techniques, particularly as used in the THE system, are I believe of great importance in themselves, irrespective of whether they are used for facilitating the construction of correctness 'proofs'.

All of this is not intended to downgrade the importance of efforts to ensure that bugs are located and removed from software, or of research efforts aimed at improving our ability to specify accurately the intended behaviour of software, and to construct correct software, and at providing rigorous proofs of software correctness. Rather, the point is that for the foreseeable future, complex computing systems must, I believe, contain effective provisions for coping with software bugs, as well as hardware failures, if such systems are to achieve really high reliability.

5. **Coping with Unreliable Components**

One obvious distinction can be made between the problems of coping with unreliable hardware, and unreliable software. This is that one would expect estimates to be available of the probability of the occurrence of the various kinds of hardware failures, based on experimental trials of prototype hardware. In contrast, predictions as to what software errors will be made must be predictions of the frailty of humans, rather than of hardware. (Incidentally, it is believed that one manufacturer has carried out an extensive survey of the kinds and numbers of errors that had been reported in a complex operating system, during several years spent modifying and extending the system — however, it was not possible to obtain access to the detailed results of this study.) In fact the situation with regard to coping with software errors is somewhat
paradoxical; in order to know exactly what precautions to take, one would like to know what errors are likely to be made. However, if one really knew this, one would take extra care in the preparation of the relevant parts of the program, in order to avoid making the errors! In practice this distinction between hardware and software is less important than one might imagine, for various reasons:

a. in many cases, one has to try and cope with an error situation without knowing whether its underlying cause is a program bug or a hardware failure — indeed in some cases one may never find out;
b. many of the precautions that one takes because of possible software errors are quite general, and not dependent on the specific type, or the location, of the error;
c. it is not always wise to rely too heavily on the accuracy of the hardware failure rate estimates. (Certainly no great claims were or are made for the accuracy or the validity of the few estimates that were obtained in our study and which are given in some of the appendices to this report.)

An important factor in the design of facilities for coping with error situations is that a complex system will, in general, be intended to produce a whole set of results, to each of which a different reliability requirement might be attached, rather than just a single result. (For example, in a system which maintains a large inventory file, inserting an incorrect value into the file may be regarded as much worse than occasionally failing to answer, or answering incorrectly, requests for information from the file). In such circumstances it is only sensible to try to design the system in such a way that its more commonly occurring errors at least do not affect the more crucial of the results that the system is producing, even though they might affect the overall reliability (and performance).

The features that are built into an operating system in an effort to cope with error situations can be divided into:

a. preparations for the possibility of errors;
b. error detection facilities;
c. error recovery facilities.
The first category includes techniques such as multiple recording of important information, e.g. file directories, the preparation of fall-back and restart facilities such as dumps, audit trails, etc., (see for example Fraser [15]), and the provision and use of protection mechanisms. This latter topic is receiving much attention at the moment, but is I am sure still at an early stage of development. One approach, involving the idea of 'capabilities', is due to Dennis and van Horn [7], and has been developed by Lampson [22] and by Yngve and Fabry, a description of whose work has been given by Wilkes [33]. The idea is that a given process (which might be part of the activity of the operating system, or arise from the execution of a user's program) should have, at any given moment, a list of 'capabilities' associated with it which indicate and delineate what the process is permitted to do. The intention is that each process be given the minimum set of capabilities that it needs in order to perform its function. If any errors are encountered, the capability mechanism limits their possible consequences, and increases their chances of being detected. The topic of protection mechanisms is closely related to that of addressing structures — if a process cannot obtain the address of an object, even accidentally, it cannot harm the object. My own opinion [27] is that this relationship has yet to be fully exploited, and that future protection mechanisms may well owe as much to work on addressing structures, such as that of the B6500 (see Hauck and Dent [14]), as to the work on the capability concept. However, it should be pointed out that the capability concept, as implemented, for example, by Lampson and his colleagues on the BCC-1 computer (see Appendix 5), is already far in advance of the protection mechanisms that are provided on most present-day computing systems.

In general, an operating system, in addition to containing its own error detection mechanisms, should be capable of dealing with reports it receives of errors that have been detected (but which cannot be dealt with) within its components, and those that have, shall we say, escaped the vigilance of the system, and have been detected outside the system, perhaps by the operators. (In fact this classification can be applied more finely, at every discernible level in the system). Its own error detection mechanisms will, ideally, only be needed for errors within the system itself — in practice they might, regrettably, have to be used for attempts at detecting errors that occur inside components, even hardware components such as processors and memory. However, as stated earlier, the aim should be that all components have reliable mechanisms for error detection, if not error correction.
All error detection is based on the provision of redundant information, whose consistency can be checked. The idea of hardware and data redundancy is well-known, but program redundancy is more novel. Clearly program redundancy is something quite different from having multiple identical copies of a program — rather it involves redundancy in the specification of the intended process. (In fact Floyd's work on program correctness proofs, described earlier, uses just such redundancy, but the consistency checks are applied before, rather than during, execution.) Examples of program redundancy, some involving redundancy already implicit in the data, others involving the deliberate introduction of explicitly redundant data, include:

a. positive checking — at a multi-way branch, where the path to be taken depends on the value of a variable, each path is taken only as the result of a positive check, leaving an extra error path to be taken if none of these checks apply;
b. sum checks — a typical example is to maintain a sum check on a table, adjusted with each change to a table entry, and checked at appropriate intervals;
c. bi-directional links — even where a uni-directional linked list would suffice, bi-directional links are used, and checks are made that an item which points at another item is itself pointed at by that item;
d. dog tags — a set of unique names are generated, and one is attached, for example, to each page of information. A process which accesses a page will do so by using its address. However, the process will also have a copy of the dog tag, which will be checked against the dog tag kept with the page, wherever it is stored. (This, and the previous technique of using bi-directional links, are used extensively in the BCC-1.)

This list is certainly not exhaustive, and more specialised techniques can often be developed, sometimes at a late stage in the design of a system, that take advantage of specialised knowledge as to the intended workings of a system (an example of this would be the 'chain-chasing' technique used in BOADICEA). Techniques such as the above form part of the folklore (but not, with few exceptions, such as Watson [32], the literature) of operating system design, or rather, of some of the more expert operating system designers, and have probably been re-invented several times. In fact the idea of dog tags was used by Eckert [12], and
the idea of using assertions for the manual checking of programs can be found in the writings of both von Neumann [16] and Turing [31].

The sorts of actions which could be considered as part of error recovery include determination of the extent of the damage, reporting of the error, and, to whatever extent possible, the repairing of the damage so that the system can continue to provide service. Determination of the extent of the damage can be implicit, from a knowledge of where the error occurred (this of course is where the protection mechanism is exceedingly useful, providing that it itself is not involved in the error), or explicit, by tentative exploration, during which facilities are exercised and consistency checks are made on data. Error repair usually involves such acts as file recovery, the re-establishment of system data structures, etc., and in the case of identifiable hardware failure, perhaps retrying the action which caused failure, or if necessary, reconfiguring the system to isolate the failed component. In the case of software error it will often be the case that all one can do is to make sure that those services which are not affected by the error are resumed with as little delay as possible.

Perhaps it is appropriate to conclude this topic by noting that these problems of error recovery are amongst the most tricky (particularly when one tries, as one should, to allow for further errors occurring during the recovery process itself) and the most important of the whole design. Indeed, one might suggest that error recovery should be amongst the first problems that are treated during the system design process, rather than, as so often happens, one of the last.

6. Generic Software

Software reliability, like performance, is essentially a 'system' problem, in that virtually every decision taken by the designers and implementors of an operating system has the potential of having a significant (and in the present state of our knowledge, often unforeseeable) effect on the overall reliability of the system. The task of obtaining high reliability from any complex operating system is extremely difficult. However, with systems such as OS/360, which might be described as 'generic' rather than specific in nature, being designed for a whole range of environments and machine configurations, the task is far worse. The different installations that use such a generic system are likely to have very different opinions as to the relative importance of the various aspects of a system's reliability, or of reliability relative to performance. This
makes it very difficult for the system designers to make rational design trade-off decisions. Thus in the case of such systems, the designers and implementors can have only an indirect (though not necessarily small) influence on the level of reliability that will be achieved at a given installation. Instead, many of the problems of achieving acceptable reliability at a particular installation will have to be tackled by the staff of that installation.

Ideally, obtaining the required trade-off between reliability and performance from a generic system would be merely a matter of choosing the right components, assigning the right values to parameters, and performing a so-called 'system generation'. Clearly, present-day reality is far from this. Very often there is a great temptation to make actual and perhaps extensive modifications to the code of a generic system — one of the major disadvantages of this course is that such modifications will have to be made to each new version of the generic operating system that the manufacturer releases, unless such 'releases' are ignored.

In view of all these factors the reliability, at least of the software, of the Barclays Bank Branch Accounting System is quite impressive. The software consists of a large suite of application programs, so-called 'middleware', and OS/360. The middleware plays an important role in coping with the problems that one might have expected from OS/360. It acts as a controlled interface between the application programs and OS/360, allowing only a trusted subset of the latter’s facilities to be used. If a new release of OS/360 has incompatibilities with its predecessor then in general only the middleware would have to be changed. (Other, more far-reaching, facilities provided by the middleware are described in Appendix 2).

7. Conclusions

A comparatively brief survey such as this can cover but a small part of the subject of highly reliable computing systems. Some topics, such as hardware error detection and correction, which are comparatively well developed, have been largely ignored in this survey in favour of less well understood and documented problems, such as software reliability. Only a small number of different environments with high reliability requirements were studied in any detail; although these are hoped to be representative of a wide range of such environments. Thus it may well be thought worthwhile for a more extensive
study to be undertaken at some future date.

The single most important conclusion to be drawn is, I believe, one that is demonstrated by several large systems, though none as well as the Project Apollo Ground Support System. This is that perhaps one of the most vital tasks to be undertaken in designing a computing system for a highly critical and complex environment is that of attempting to minimise the extent to which reliance is put on the correct and continuous functioning of the system. However, it is also clear that the problems of achieving high reliability, at acceptable performance and cost levels, are worthy of much further study, although one would be naive to expect quick solutions.

8. Acknowledgements

It is a pleasure to acknowledge the assistance that we have had from many different sources in preparing this report. In particular, we wish to thank Mr. J. W. Lewis and Mr. M. R. Dillingham of B.O.A.C., Mr. A. Duncan, Mr. Cainer, and their colleagues at Barclays Bank, Mr. D. A. Weir and his colleagues at Standard Telecommunication Laboratories Ltd., Mr. E. A. Newman, Mr. D. O. Clayden, and Mr. M. A. Wright, of the National Physical Laboratory, Dr. P. Cliffe of Westminster Hospital, Mr. M. D. Oestreich of Computer Analysts and Programmers Ltd., and Dr. B. W. Lampson of the Berkeley Computer Corporation.

The present report is the result of work by Dr. J. Eve, Dr. A. Mascall, Professor W. C. Lynch and the author. In particular, Appendices 2 and 3 were taken from accounts prepared by Dr. Mascall.

Preparation of the report was aided by a grant from the Science Research Council.

9. References

[6] Dieterich, E. J. and Kaye, L. C.  A Compatible Airborne Multi-
[7] Dennis, J. B. and Van Horn, E. C.  Programming Semantics for Multi-
Moore School of Electrical Engineering, Univ. of Pennsylvania,
pp.19-32.
for an Electronic Computing Instrument, Part 2, Vol. 1.  Institute for
pp.80-151).
Software Engineering  (Eds. Naur, P. and Randell, B.) NATO Science


APPENDIX I

The BOAC On-line Airline Reservations System

Overview

The main computing power at the BOADICEA centre consists of three IBM 360/65s, each with 512K bytes, with manually switchable connections to a bank of eight 2314s, and a large number of tape decks. There is also a Model 40, and a fourth 65 is on order. There are three main categories of use:

1) Batch processing.
2) System development, in particular for the airline reservations system.
3) Airline reservations and departure control.

One or two of the batch processing jobs are time critical, notably the flight planning system, which is run three times a day and uses about 15 hours of CPU time per week. Flight planning is based on information obtained from weather forecasters (Bracknell) and flight control (Prestwick). The reliability of the flight planning system is much higher than that of the partly-automatic partly-manual communications network between London, Bracknell and Prestwick. Although computer hardware reliability is not as high as one might reasonably hope, there is sufficient fallback capability to make up for this, though other batch work, and especially development work, sometimes suffers as a consequence.

The most interesting system, from the point of view of reliability, is the BOADICEA reservations system. The system is run on a single 65 connected into a low-speed and a medium speed communications network. The software has been developed collaboratively by BOAC and IBM, and BOAC is now developing software to sell to other airlines. This software development, together with that involved in extending the present airline reservations and departure control system involve considerable computer utilisation, and is a major reason for obtaining a fourth Model 65.

Communications Network

The BOADICEA system is connected into a low-speed (150 baud) network (The Commonwealth Airlines Network), which involves message switching computers at London, Hong Kong, Sydney and New York. This network also has links to other similar networks in Europe and the United States. The network is used for transmitting information to outlying BOAC offices and to other airlines. Some 80% of the traffic concerns reservations, the rest concerns aircraft movements, and general administrative messages.
The network serves some 370 teletypes, and carries about 175,000 messages a day, the average message length being of the order of 100 bytes. The network in general provides multiple paths between nodes. Line reliability is of the order of 95-99%. There is enough security and checking in the network for unreliability to manifest itself almost entirely in the form of delays to messages, rather than corruption of messages.

The main communications facilities use the medium speed lines (2400 bauds), of which there are eight. Each line leads to a TCU (terminal control unit). These TCUs are situated at various BOAC locations, and are connected to printers and agent sets (character display terminals). There are about a thousand agent sets connected. The average message rate is three per second, the peak rate is 15 per second. Message lengths are on average approximately 60 characters for input, and 110 characters for output. Messages are blocked together by the TCUs.

The medium speed lines are used for airline reservations messages to and from BOAC locations (other than small outlying locations) and for departure control. This latter is much more time critical than reservations, since trouble with departure control can lead to flight delays. This network also provides multiple paths, and includes two transatlantic lines (in separate cables).

The following table summarises system reliability during a fairly typical week, shortly prior to our visit.

<table>
<thead>
<tr>
<th>Reliability</th>
<th>Availability</th>
<th>Number of Outages</th>
<th>Total Mins. Lost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% of planned</td>
<td>&lt;2min.</td>
<td>2-5min.</td>
</tr>
<tr>
<td>CPU</td>
<td>99.3</td>
<td>4.0</td>
<td>13.0</td>
</tr>
<tr>
<td>Av. per TCU</td>
<td>98.5</td>
<td>5.0</td>
<td>2.2</td>
</tr>
<tr>
<td>Av. per CRT</td>
<td>9.0</td>
<td>15.2</td>
<td>3.3</td>
</tr>
</tbody>
</table>

In the table CPU availability includes both hardware and software aspects. During this week, a typical agent set could not be used for a total of 147 minutes — approximately a half of which was due to the communications system.

Main System

The reservations system has been in operation for two years. The three basic parts are ACP (Airlines Control Program), which was produced by IBM, IPARS (International Programmed Airline Reservations System), which was
produced collaboratively by BOAC and IBM and the Departure Control System, which was produced by BOAC.

The IPARS program was based on the PARS program, produced by IBM for the domestic American airlines. This program and ACP and the team that developed them can trace their origins back to the original SABRE system, produced by American Airlines and IBM, for a twin 7094 installation. Thus the present BOADICEA system, which consists of approximately 500,000 instructions, represents the culmination of over ten years development. Further development of BOADICEA now under way includes hotel reservations and fare calculations (BOAC) and message switching (IBM).

The program is now changed about once a week. Before being incorporated changes are tested in a miniature version of the system, which is run for about two hours a day on standby equipment. The original system was tested for about six months using about 100,000 simulated messages. All programming is done in assembly language (as opposed to BOAC's batch processing which is all done in PL/1).

The basic system structure is very reminiscent of the original SABRE system. The workload is regarded as being made up of very many small independent tasks, most of which access and/or modify the set of very large common files. The whole system is modular, and all program modules fit into 1055 bytes.

One of the most important aspects of obtaining reliable operation is that of detecting errors, preferably before too much damage can ensue. One of the techniques is to make positive checking a standard programming style — for example, if a branch has to be made to three different labels, depending on whether a variable has the value zero, one or two, this will be done with three tests, so that if the variable has some other value, an error report will be made (this technique, incidentally, was also used to considerable advantage in the Whetstone KDP9 Algol Compiler!) The errors reported by such techniques can be due to either hardware or software — BOAC's experience has been that by far the majority come from faulty software. Clearly, judgement has to be exercised as to the extent of positive checking — there was a considerable amount in the original design of IBM's ACP, and although many were put into IPARS originally, many more have been added, as operational experience has been gained.

A more explicit error detection technique that has been found very useful is what has become known as 'chain-chasing'. Chain-chasing
involves tracing through the many and varied linked lists kept in core storage to check their consistency, and that all pointers point to the correct type of data. This is now done at each exit from each system macro, i.e. very many times per second, and in general allows one to identify immediately which program has given rise to the error. Chain-chasing uses about 20% of the CPU time, but this is acceptable, at least at present, since the CPU is very lightly loaded (CPU utilisation is only about 25-30%).

A similar technique called 'recoup' is used to check all the files held on disk storage. This process involves tracing through all the linked lists, creating a new set of directory records, and freeing dead records. The dead records which are found by recoup will include those that have arisen from, for example, partially completed tasks which hit an error condition and had to be terminated. Recoup is done two or three times a week, while the system is operational, although the system has to be stopped momentarily. It requires about three hours of elapsed time, since it involves accessing every record, perhaps many times, but much of the work can be done on one of the off-line computers. It has proved to be extremely valuable, because of the way it assists in the vital task of making sure that the main files do not get out of control.

(Perhaps equally important for avoiding corruption of files is data vetting. Very extensive checks are made on all input data before any action is taken on the basis of it.)

Error Recovery

In order to achieve acceptable service it is vital that, to the greatest extent possible, the system continues after an error has been detected. Thus, if a condition arises which prevents completion of processing of an input record, wherever possible a detailed error message (rather than a standard dump) is printed, and the system ploughs on.

All transactions which change a file are logged onto magnetic tape. Transactions are blocked together in groups of 80, so that up to 80 can be lost when the system crashes. In addition a 'capture', i.e. complete incremental dump of all files, is made every other day. If an inventory record is lost, the previously dumped version can be re-read from the dump tape. This takes about 30 minutes, during which time there will probably be little action against the record, and all of which action will be logged.
A considerable amount of information (e.g. programs, basic flight inventory records, but not the passenger name records) is recorded in duplicate, on separate disks, mainly as a safeguard against hardware, rather than software errors.

In the case of hardware errors it is sometimes necessary to reconfigure the equipment, and to bring up the system on the stand-by computer. This can be done in a matter of a few minutes, though will of course cause disruption to the off-line work that in all probability will at that time be in progress on the stand-by computer. Currently the system has to be restarted, following an error, on average once or twice per day. On average about 10% of these restarts were known to be due to hardware, the rest were due to software or operator error. In a recent survey, it was found that 16% of the unplanned downtime could be attributed to hardware failures, the cause of the remaining downtime was due to software or operator errors, or was unknown. The difficulty of deciding whether an error was due to hardware or software could be very troublesome. There was the tendency now to assume that trouble was due to software — on occasion this had resulted in poor service over a lengthy period of time, which could have been avoided, if it had been realised that operation should have been switched to the stand-by computer.

System Reliability

In retrospect it was felt that the original concern about reliability was perhaps slightly excessive, although this opinion might well be coloured by the fact that recently the system had been going very smoothly.

For example, the extensive duplication of records on disk, intended as a safeguard against hardware error, now did not seem altogether justifiable, and it would probably be sufficient just to duplicate all the program files, and the in-core records needed for system restart. The occasional loss of an inventory record on disk has turned out to be not so serious as feared, because of the logging and dumping facilities, which nearly always enables the record to be re-created. (The very occasional total loss, e.g. because of logging information lost at a system crash, is not quite so serious as one might expect, since in any case on average there are 4% 'no-shows', i.e. people booked on a flight who do not show up for it!).

22
However, it would be difficult to exaggerate the contribution of the programming techniques of positive checking, and of ploughing on after an error, together with the very extensive back-up hardware, whose cost could be justified by its use for batch processing work, to achieving satisfactory operation.
APPENDIX 2

Barclays Branch Accounting System

Greater London Computer Centre

The Greater London Computer Centre houses three IBM 360/65s and one 360/50. These machines provide a branch accounting service, a clearing service for sterling accounts of the chief foreign branches in the U.K., an on-line foreign mail payment order service, payroll accounting and a standing order payment service. By far the largest volume of work is the branch accounting service. The basic system used is IBM OS/360 running in an MFT environment.

During the working day a teleprocessing system is running which acts as an on-line data collection network. The individual branches prepare punched paper tape records of transactions received at the branch, and transmit them to the centre via 3940 paper-tape terminals over leased phone lines. About $10^6$ entries per day are made from all the branches, and the system holds information on about 1.75 million individual accounts. Normally each of the three 360/65s is servicing one teleprocessing network, but in the case of hardware failure one CPU can service two of the networks at the same time. In fact there is a large proportion of idle time, since most of the time is simply spent in polling the lines. Thus, if the system is down for as much as two hours early in the day it is still possible to be on par at the end of the day. In the Centre itself there is sufficient standby hardware, both in processor time and in I/O facilities, the only serious source of backlog being at the branches, where a failure of the paper-tape machines will not be detected immediately and can result in a lot of hand keying.

Running concurrently with the data-collection system is a separate teleprocessing system using IBM 2260 display consoles for on-line processing of foreign mail payment orders into and out of the U.K. Input is received from 40 operators working in the foreign branch in Central London. In contrast to the data collection system, this service demands a much higher standard of availability since the operators need to be kept busy. In addition, although the number of entries is very small compared with those from all the branches, very large sums of money are involved, to the extent that overnight interest becomes important so that payments have to be effected rapidly.
Branch Accounting

At the end of the working day the system is switched over to the overnight batch processing of the branch accounts and standing orders. The main feature of this work is that it is extremely time-critical. All the branches have to be processed by the system, the records updated for each branch, and the listings prepared and posted to the branches in time for the following day's work. This must be completed in a maximum of 20 hours machine time on the two active 360/65s, i.e. 10 hours per CPU, with one CPU on standby. The 360/50 is also used as part of this batch-processing complex, so that the complete division of work is roughly: 2½ CPUs for branch accounting, ½ for standing orders, and 1 CPU on maintenance and software testing.

At the start of the overnight session the entries received during the day on the data-collection system are sorted into lists to be dealt with by each of the CPUs. The entries are only roughly checked for format at the time of collection, and some further checking is done at this stage. Once the updating of the individual branches has been started the operators are responsible for the mounting of the disks (two per branch) in the correct sequence. Normally a strict rota for the ordering of the branches is kept. This system has now been running for nearly two years, and its organisation is considered to be well established with regard to operating procedures, so that such catastrophes as incorrect mounting of disks, resulting in loss of data-sets and the further loss of time spent in recovery, are rare. However, in that period the workload has built up considerably, so that there are now only about 14 hours idle time in 400 hours of operation. This lack of standby time results in a loss of time available for testing programs in the event of a system failure.

Software

The initial version of the Branch Accounting system (Phase A) has been operating for nearly two years. A major change to a new system (Phase B) is planned for about April 1971, after the changeover to the decimal currency has taken place. At present about two small changes per week are made to the system. The constant development of the system to date has been in order to achieve more speed, to service more branches and to provide more services to the branches. The programming languages used for the applications programs are PL/1 and COBOL. Programmers are recruited to a large extent internally, but also directly amongst graduates.
and school-leavers. They are given an initial three months training followed by on-the-job training. Programmers work in shifts so that someone is on call by the operators to deal with software failures.

At the outset system security was considered to be a major concern in its design. This is much more a problem of preventing unauthorized access to files, rather than one of security against fraud. To this end, the bank's internal auditors and managers of the computing department closely control the system design and standards. Another security is that of making snap checks between the master copy of a program and that which is running. Nowadays this is thought to be much less of a problem, and the above procedure quite adequate. It is, however, realised that should a responsible programmer suddenly have a grudge against the bank he could cause serious trouble with the system before being found out.

**Middleware**

One feature of the Branch Accounting system which is important for the preservation of file integrity is the 'middleware' that was written by CAP under a contract to Barclays. As its name implies, it is an interface between the individual job programs and the OS/360 operating system, principally the data management functions of OS/360. It was designed to control the access of individual data-sets and to simplify their use by job programs. The main aspect of this is that during the updating of a branch file the allocation of the old and new master files is controlled by the middleware. Even if an application programmer wants to do an 'update in place' there is no means for him to do so. When the update has been completed to the satisfaction of the job program the new master may be checked for correctness and only then is it automatically designated by the middleware as the latest master copy. In case of a system failure at any of the intermediate stages the middleware will ensure that the job is completely re-run from the proper copies. Fallback to previous old master copies is also controlled by the middleware.

In all, the reliability of the software for the Branch Accounting is considered very satisfactory. In particular, the CAP middleware has fulfilled its original specifications, and has proved to be very reliable. There have only been 71 recorded errors in the production versions of the Branch Accounting programs in over 18 months. In a series of three weeks shortly before our visit the following figures
were quoted for the total number of hours downtime per week attributable to different causes:

<table>
<thead>
<tr>
<th></th>
<th>Branches</th>
<th>OS/360 Software</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>25</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>Week 2</td>
<td>25</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Week 3</td>
<td>26</td>
<td>1\frac{1}{2}</td>
<td>11</td>
</tr>
</tbody>
</table>

In addition, another 12-15 hours per week downtime are due to I/O troubles of various nature. If so, the total time out is now somewhat less than 20% of the available time. The above figures do not account for time lost when an automatic recovery has been performed, resulting in the re-running of a job. The bank conclude that although the workload is now large enough to cause curtailment of some lower priority work, such as testing and payroll, in the event of a prolonged failure, the capacity of standby is sufficient to cope with present demands, and the main source of hold-up is hardware failure of one sort or another.
Overview

The system which is presently in operation is based on two twin-CPU B5500s, but is due to be replaced by an entirely new system on two twin-CPU B6500s. It is an on-line random updating system fed by Burroughs TC500 terminals, which are installed at local branches throughout the country. Files for each branch that has access to the system are held on Burroughs B475 fixed-head disks, and back-up is provided by Burroughs B423 magnetic tape units. Of the two systems, the first services about 20 large branches, the second about 60 smaller branches (of which the Clayton Street Branch, Newcastle upon Tyne, is one). The machine configuration on both is the maximum possible for the B5500: 32K words main memory, 32K word extended core memory (ECM), two line printers and two card readers. The machine servicing the smaller branches has 20 disk modules (capacity of each 9.6 megabytes) and eight tape units.

The system is intended to provide on-line access of all records for accounts held at a branch throughout the day for updating, reference and control purposes. Overnight, the records are updated with information from other sources received on tape at the computing centre. In the smaller branches, such as Clayton Street, the TC500 terminals provide the access to the system, and in the large branches these are supplemented by papertape readers for batched input using tapes produced at the branch.

At the Clayton Street branch there are two TC500s with a third as a backup, which is normally used as a desk accounting machine. At the start of the day, 08.00 hours, when the system comes on-line, one of the TC500s is put on to report on the previous day's transactions. This produces the overnight report on the status of all accounts referenced during that day, showing the new balance for each account. It is aimed to complete this report by 10.00 hours, so that the latest credit balances are available at the counter when the doors open. The second terminal is then started, and checks received from the morning mail are keyed in by the operator. At the same time the other terminal is re-set in order to effect routine enquiries for individual accounts, obtain detailed statements where desired, enter information on stopped checks, overdrawn accounts, etc. The majority of the work is thus batched, the incidence of random enquiries being very low. Detailed statements for an individual account are not normally obtained, and the main output is the
overnight report for the counter ledger.

The system holds information on current accounts (56%), deposit accounts (30%), loan accounts (7%) and group accounts (2%), plus sundry accounts used for branch housekeeping. The response time for a simple enquiry, such as the correct balance of a given account, varies from 30 seconds to five minutes, depending on system load and the availability of the line. Roughly 100 different enquiries are possible, and in addition the programming of the TC500 makes its routine operation very simple. The principal benefits are the avoidance of lengthy ledger searching for simple enquiries, the speed with which vital action on particular transactions or accounts could be taken, and the much reduced paper-work. The system may be used as desired by the Branch Manager, although obviously the recommendations of the systems people given at training courses will greatly influence their decisions. At Newcastle the daily report is considered the most important item, followed by the batched entries of checks received at the counter. In this respect the use tends to correspond to a remote batch terminal. At Newcastle the branch produces about 80 feet of waste input, input that is not retained, and 40 feet of records on the TC500s each day. This corresponds roughly to about 200,000 characters per day on that line.

**Software on the B5500 System**

The Barclays system runs under control of the Burroughs B5500 MCP (Master Control Program). This module is a standard Burroughs product, which controls all access to I/O devices, allocation of memory, and scheduling of the processors. The central component of the system is the Executive, which is started by the operator during a start-up or restart, and which controls the start-up and close-down of all other programs. The more important of the set of programs controlled by the executive are as follows:

1. Data communications - Handles the lines to TC500s.
2. I/O Module - Handles disk and tape I/O.
3. Monitor - Provides operator communication with the executive.
4. Trans Q dump - Periodically dumps the current contents of the transactions file to enable a restart.
5. Update - Applies the entries, either from the clearing or the data-comm system.
6. Inquiries - Handles inquiries, and also treats urgent amendments to the main file.
7. Refers — Reports on overdrawn accounts.
8. Reports — Produces certain reports for branches.

The first four of these modules are written in Algol, the others being written in COBOL. The Executive is written in Algol, and the MCP itself, which was written by Burroughs in ESPOLE, is not normally touched by the people at Barclays.

The B5500 system is only an interim measure, and it was never expected that it would provide a service which would be as complete or as reliable as that expected from the B6500s. However, due to delays in the delivery of the B6500s, the B5500 system has had to be used for a much longer time, and much more extensively, than was originally planned. The experience gained with the present system has obviously been very useful, and the application programs already developed will be used in the B6500 system, although there is a divergence of opinion on the amount of effort involved in re-programming for the B6500 system.

When it was decided to go ahead with this interim B5500 system, the specification laid down by the bank was that it should follow as closely as possible the specifications given originally by Burroughs for their projected B8500 banking system; i.e. the B8500 system should be 'tailored' to fit into a B5500. The hardware of the B5500 series imposes the considerable constraint that programs are non-modular. This makes the writing, check-out and subsequent modification of all programs much more difficult.

The workforce at Barclays has been organised as follows. At the management level, the principal responsibilities are the specification of overall policy and the general quality of the staff. The Project Controllers take care of the scheduling of various projects, along with the co-ordination of the Systems Analysts and the Programmers working in the different teams. The team leaders are usually Systems Analysts, and they are the people responsible for the quality of the code produced by the team. It is general policy that the programmers have a good knowledge of the hardware and system architecture. This is brought about partly during training, but more importantly by the programmers being organised in shifts so that a member of the team is on call by the operators at all times of the day in order that he may both appreciate the disruption caused to the system by a failing program, and be on hand to diagnose and correct the fault.
The programming language used for writing the applications programs is COBOL. Although this language is rather less efficient than the Burroughs ALGOL, both in terms of compiler efficiency and generated code, it is preferred, since it is more relevant for the type of data management required by the system, and it is more easily learnt by the applications programmers. To the present time the check-out of modules before final incorporation into the system has been done by the system of 'vetting' by the team leaders, plus runs on a test system during times when the machine is available. If a new program fails during production, a standard fallback procedure to the previous version is taken. A recent feature, shortly to be introduced, is the addition of a 'tracing facility' in the Burroughs COBOL compiler (a Burroughs product). It is hoped that this will enable the programmers to locate the most heavily-used parts of certain programs, and enable them to optimize the performance in these sections. The general feeling among the senior programmers is that this will prove very enlightening about the reasons for poor system performance when it is implemented.

System Reliability

Although file integrity is the most important aspect of a banking system, it is not as critical as might first be feared by the bank's customers (for instance), in that the original records on paper of the transactions demanded are always available at the branch. For the same reason, the appearance of 'criminal programs' in the system would not benefit the perpetrator in financial terms, even if such programs got past the normal vetting procedure and system management watchdogs. File integrity and full backup is, however, necessary in order to achieve continuous on-line service to the branches. In terms of a system crash, when the files are still intact, it is necessary to update the files at the restart with the queue of transactions which had been received prior to the failure, but not applied. In terms of a disk crash, resulting in a corrupt disk, the disk must first be restored from the most recent dump tape, followed by an updating from the transactions received after that dump. Disk failures due to hardware faults are unfortunately frequent, and result in quite long interruptions in service during the lengthy restore process. Occasionally, where the failure is limited in extent, service may be resumed for most branches, but certain may have to be denied access for a whole day. Two security tapes are used to hold the above restart information, both of which are in duplicate. The 'old copy tape' contains disk records that are read into core for updating,
buffers from the line network, and in-core tables dumped at regular intervals. The 'new copy tape' contains updated disk records as written back to disk, identified in the same way as the records on the old copy tape. These tapes are held for a number of days, since it is from these along with a dump tape that a disk module must be reconstructed to an up-to-date state. A normal restart, due to memory parity failure for example, takes about ten minutes. Transactions received from the branches are not applied immediately, but are written to a transactions file and subsequently applied to the main file by the UPDATE program.

The performance of the MCP appears to be very satisfactory, and there is good cooperation with Burroughs on putting right any faults. Since the addition of ECM some three months ago, the performance of the operating system under heavy load has increased considerably. Two particular faults are degradation due to memory fragmentation, and occasional system failure due to loss of interlock between the two processors. On rare occasions patches have been applied to the MCP, but these are rapidly dealt with by Burroughs at the source. The MCP concerns itself only with the allocation of real devices: allocation of files needed by different programs is done by the I/O MODULE, and a simple guard against a deadlock situation caused by file contention between only two programs has been incorporated here. Deadlocks do occasionally occur, and cause a restart but this is tolerated and is thought to be rare enough to be ignored. In particular, the subsequent restart normally bypasses the deadlock.

The MCP system makes no attempt to recover from serious hardware failures. File integrity is provided by fairly standard techniques of duplicating files, use of audit trail tapes, and recovery from old master copies. One difficulty with the provision of duplicate files is that the application programmer has no means of informing the MCP that the files are duplicates, and hence should perhaps be kept on separate disks, and has no means of finding out whether a file has indeed reached the disk. This is one area in which the Burroughs software, though much more convenient to use than OS/360, is deficient.

It appears that the data communications module has proved to be the most reliable part of the system, and at the present time gives no trouble at all, whereas faults are frequently found in the other applications programs. The data communications module was written in Algol, but large use was made of 'stream procedures', which amounts to writing in a low-level language. Initially it was developed from a module
supplied by Burroughs, but in the process was almost completely re-written. Its reliability seems to be due mainly to the high standard of the two people involved in its development, and the ample time given for checkout. In addition, the documentation provided by Burroughs on the TC500s was complete, and of high quality. The errors encountered in the applications programs arise mainly because insufficient time is now available for testing off-line, and because they have to be modified quite often, both for exceptional circumstances, and in the course of adding new features. The non-modularity imposed by the system on the programs contributes to the difficulty of achieving error-free modifications.

Operator action on the B5500

The communication with the operator and verification of operators' actions as provided by the Burroughs MCP have proved to be very satisfactory particularly compared to IBM OS/360 used for the Branch Accounting System. The MCP keeps the operator informed of the system status as different tasks are started up and stopped, signals error conditions that are detected in the applications programs, and makes requests on the operators for mountable volumes (tapes). Only messages requiring operator action are issued: a system log is also kept, which may be printed at the end of the day. Tapes are recognized by volume name, and the system will search the available tape drives for the correct label when it requires it to be mounted, so that there is very little room for operator error in the mounting of the audit trail tapes.

In the case of a failure in an applications program, the operator will call on the programmer responsible to deal with the error, and standard fallback procedures are taken if necessary. For recognisable system errors the operator can take the correct restart procedure, and in the case of an unrecognisable situation he must call the chief operator. Naturally, ad hoc remedies cannot be tolerated, and for any serious errors a senior member of staff must be called. Although in normal running the MCP dictates to a 'slave' operator, recovery procedures have to be initiated manually, and there is a fair amount of operator initiative required. However, his permissible actions are very well-defined by the 'book of rules'.

Changover to the B6500 system

A pilot system is already running on the B6500 under test conditions, but its date of commission has been put back due to the considerable amount of work required on the present systems for the changeover.
to decimal currency. The Burroughs MCP for the B6500 will be used as the central part of this system. Communication with the remote terminals on the B6500 is now effected through a peripheral processor, the Burroughs DCP (Data Communications Processor). The work on the terminal support, both at the level of the DCP and in the B6500 is already well advanced, and has been done by the programmers responsible for the Data Communications package in the B5500. In the B6500 system the communication with the DCP is made via a Data Communications program, which in turn interfaces with a message processor (MCS : Message Control System). This arrangement will make failures in a particular applications program much more transparent to the user, since messages from terminals can continue to be received and stored while the recovery of the failing program is undertaken. (It should be noted that in the present B5500 system terminal communication has also to be suspended during the intervals that the history dump is being written to the audit trail, resulting in an appreciable delay at the terminals every ten minutes or so). The applications programs for the B6500 are being taken over from the B5500 system, although it appears that a considerable amount of re-programming is being done. In addition, new facilities are being added to the present suite of programs. While the system is working as a pilot scheme there is still plenty of time to enable checkout of the individual components, but it is clear that with the workload envisaged for the B6500, program checkout will be a considerable problem, and may well prove to be a source of difficulty both for maintaining and extending the system.
APPENDIX 4

The STL Dependable Process Computer

The Dependable Process Computer was built in order to test the efficacy of various redundancy techniques, aimed at ensuring continuous ultra-high reliability. It has been described briefly by Darton [5]. Since the details of the project are company confidential, this brief account largely confines itself to points given in Darton's paper.

The computer uses redundancy at a fairly low logic level in order to detect and identify hardware faults (whether transient or solid) and allow computation to proceed whilst the fault is repaired. Since faults are in general identified down to the logic board concerned, the time needed to repair the computer is merely that of replacing the indicated board. The computer is divided into several different areas, each of which is checked separately. A single fault in each area (including in any checking or voting circuits) can be tolerated, without having to suspend service.

Various different redundancy techniques are used in the different areas of the machine. In the memory and register areas, a 5-bit Hamming code and an overall parity bit are associated with each 16-bit word. The Hamming bits are sufficient to detect any single bit error in the data word — the parity bit can be used to check whether a reported error is indeed in the data word, or whether it is in the circuits which check the Hamming code. The parity checking circuits are themselves duplicated, and a parity error is assumed only if both checking circuits report it. In the control unit and functional unit triplication and majority voting on the level of single bits are used.

However, due to the size of the memory, relative to the rest of the computer, the average level of redundancy is just under 100%. This is of course a great improvement over the level that would have been required for triplication of the entire computer, plus majority voting. In addition, the chosen scheme gives a much better indication of the location of any fault.

The computer has achieved a total of 22,000 hours of operation with no unscheduled downtime. (Downtime has occurred only on occasions such as when all electrical power to the laboratory in which the computer is situated has been switched off for maintenance purposes.) During this time 22 errors were detected and repaired merely by replacing...
the indicated circuit board, without interfering with the operation of the computer in any way. There has been no evidence that any mistakes have been made in any calculations or controls which have been performed.

Surprisingly enough the machine has proved, to date, a slight disappointment. The reason for this is that it had been developed with the process control field in mind, and so far it would appear that the reliability of the computer is much higher than what is needed in this field. To date most digital process control systems have been built on top of 'classical' analogue control devices which can keep the process under control, sometimes even for hours, though perhaps working at reduced efficiency, when the digital system fails. However, recent developments have indicated a move towards all-digital control, in which case the need for high reliability and availability from digital process control computers will increase dramatically.
APPENDIX 5
The BCC-1 System

Overview
The BCC-1 System is being developed by Berkeley Computer Corporation, a small company whose personnel, whilst at the University of California, Berkeley, designed and built the XDS 940 time-sharing system, and the CAL time sharing system (on a CDC 6400) and participated in the design of the SCC-6700 system. At the time of writing the present status and future prospects of the BCC-1 are unclear. The system was recently reported to be near to being operational, having been in use for some time for software development, but the financial position of the Berkeley Computer Corporation was, to say the least, shaky.

The corporation was formed in order that the design team could pursue their ideas as to system architecture for multi-access computers. It proved impossible to continue their work at the University of California in view of the amount of hardware development involved. Instead, financial backing was obtained, on the basis that their initial system design would be used to provide computing service to a set of independent time-sharing vendors. The intention was that the BCC-1 should service 500 terminals. Since separate (and competing) time-sharing vendors would be retailing computing time on the BCC-1, and developing their own subsystems, the designers put a lot of their effort into the problems of security.

System Configuration
The BCC-1 system incorporates two CPUs, core memory, and a large fast paging drum, as well as disk storage, peripherals, etc. In addition three specialised microprogrammed processors are used for various resource management tasks such as disk scheduling, processor scheduling, and device management. (see Watson [32]).

The intention in later models was to have a considerable amount of redundancy, at the level of large functional boxes, such as processors, memories, etc. In the present system, each specialised microprocessor is connected only to those other modules that it needs access to. In a future model it would be hoped to have writeable control stores for each microprocessor, and sufficient connections that any microprocessor could if necessary take over the work of another one. The purpose of the microprocessors, whose instruction rate is comparable
to a CDC 6600, is to free the main processors from much of the work involved in conventional operating systems. For example, one microprocessor implements a very sophisticated sector queueing scheme on the drum so that in general a job can be swapped into memory within a single drum rotation. Another processor relieves the main processors from any concern with interrupts.

The Protection Mechanism

The basic supervisor (which unfortunately consumed 20,000 instructions) provides a protection mechanism, based on the concept of capabilities, it being possible to have capabilities relating to files, processes, interrupts, data, programs, devices, etc. This mechanism is a development of that described in reasonable detail by Lampson [27], and hence will not be described here. It is understood that the protection scheme has provided quite adequate protection — it has for example allowed different versions of different subsystems and emulators to be run concurrently, with complete protection of each from the consequences of errors in the others. Furthermore, it is possible for one program to pass arguments to, and receive results back from, another program, without either program being able to access any other data belonging to the other. However, the relationship of the protection mechanism to the address space provided to programmers has given problems, and it is realised that this area is worthy of further development.

Reliability of the Basic Supervisor

Within the basic supervisor it is of course not possible to take advantage of the protection mechanism. In partial compensation several redundancy techniques are used, including:

1) Singly-linked lists are avoided, and doubly-linked lists are used instead.

2) Pages of information are identified by both a unique name and an address. Names are 48-bit integers, generated in sequence, throughout the life of the system. (It is inconceivable that this would involve using all \(2^{48}\) possible names.) Access to a page is by address, but the name of the page must already be held by the process which is trying to fetch the page, so that it can be checked against the name held with the page.

3) Much information is held redundantly, and this redundancy is checked at appropriate times. For example, each process
has associated with it a list of the resources that it owns, and each resource has associated with it information about who owns it.

**Error Recovery**

The basic philosophy of error recovery is, at the point the error becomes apparent, to stop everybody, copy the contents of core onto drum and then investigate what level of restart is necessary. Restart can be thought of as building a new system from the wreckage of the old, transferring across any good information, and perhaps bad information (with tags to indicate this) from the old into the new system.

The consistency of core and drum tables is checked – if core is lost, it is unfortunately necessary to reorganise the drum contents as well, since it is not known which of possibly many copies of a page on drum is the latest one. (Sequence numbers, which would solve this problem, have yet to be kept with pages on the drum). The global table, which describes active processes, is regenerated from the various context blocks, (held either in core or on drum). If the contents of the drum are lost, then it is necessary to regenerate the contents of the disk from magnetic tape; however, if the drum contents, but not core contents are lost, it is known what was on the drum, and therefore which pages on the disk have to be rolled back into the drum, or tagged as suspect.