Software Fault Tolerance: An Evaluation

T. Anderson
P. A. Barrett
D. N. Halliwell
M. R. Moulding

Abstract

In order to assess the effectiveness of software fault tolerance techniques for enhancing the reliability of practical systems, a major experimental project has been conducted at the University of Newcastle upon Tyne. Techniques were developed for, and applied to, a realistic implementation of a real-time system (a naval command and control system). Reliability data was collected by operating this system in a simulated tactical environment for a variety of action scenarios. This paper provides an overview of the project, and presents the results of three phases of experimentation. An analysis of these results shows that the software fault tolerance approach can be successfully utilised in critical real-time systems to achieve a significant improvement in system reliability.

Series Editor: M. J. Elphick

© 1985 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

SOFTWARE FAULT TOLERANCE: AN EVALUATION.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 202.)

Added entries
ANDERSON, Thomas.
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract
In order to assess the effectiveness of software fault tolerance techniques for enhancing the reliability of practical systems, a major experimental project has been conducted at the University of Newcastle upon Tyne. Techniques were developed for, and applied to, a realistic implementation of a real-time system (a naval command and control system). Reliability data was collected by operating this system in a simulated tactical environment for a variety of action scenarios. This paper provides an overview of the project, and presents the results of three phases of experimentation. An analysis of these results shows that the software fault tolerance approach can be successfully utilised in critical real-time systems to achieve a significant improvement in system reliability.

About the author
Dr. T. Anderson is currently a lecturer in the Computing Laboratory at the University of Newcastle upon Tyne.

Suggested keywords
CONCURRENCE
EXPERIMENTATION
REAL-TIME SYSTEMS
SOFTWARE FAULT TOLERANCE
SOFTWARE RELIABILITY

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 001.64404
U.D.C. 681.322.06
Software Fault Tolerance: An Evaluation

by: T. Anderson
P. A. Barrett
D. N. Halliwell
M. R. Moulding

Centre for Software Reliability
University of Newcastle upon Tyne

March 1985

Abstract

In order to assess the effectiveness of software fault tolerance techniques for enhancing the reliability of practical systems, a major experimental project has been conducted at the University of Newcastle upon Tyne. Techniques were developed for, and applied to, a realistic implementation of a real-time system (a naval command and control system). Reliability data was collected by operating this system in a simulated tactical environment for a variety of action scenarios. This paper provides an overview of the project, and presents the results of three phases of experimentation. An analysis of these results shows that the software fault tolerance approach can be successfully utilised in critical real-time systems to achieve a significant improvement in system reliability.

Submitted to IEEE Transactions on Software Engineering
(Special Issue on Software Reliability; editor A. Goel)
1. Introduction

The process of software development is usually described in terms of a progression from user requirements to the final code, passing through intermediate stages such as specification, design and validation. Of course, progress through these stages is rarely unidirectional, and "final code" must be considered to be a misnomer given the demand for subsequent software maintenance. An engineering approach to software development should enable software to be produced on time, within budget, and in accordance with user requirements. One important aspect of these requirements concerns the reliability of the software. Software reliability requirements can be expressed in a number of ways, of which the simplest, perhaps, is to impose an upper limit on the measured rate of failure over a specified interval.

Given that reliability criteria can (and should) be imposed on software systems, how can these standards of reliability be achieved? Fortunately there is a wide range of techniques available to the software developer, all intended to enhance software reliability. These techniques may be categorised as follows [4]:

1. Techniques to avoid making mistakes - such as design methodologies and notations - referred to as fault avoidance.

2. Techniques to find and remove mistakes - such as design reviews, code inspection, program analysis, testing, verification, all followed by debugging or redesign - referred to as fault removal.

3. Techniques to cope with mistakes - defensive programming based on redundancy - referred to as fault tolerance.

The major obstacle impeding the construction of reliable software according to engineering principles is the shortage of data on the effectiveness of these various techniques. Fault tolerance techniques have played a major role in the development of reliable hardware systems [11,16], but have been much less widely used to cope with the possibly more serious problem of software reliability. Over the last ten years there has been considerable research activity addressing a range of issues in the field of software fault tolerance (see, for example, references 1, 4, 8, 26 and 27). One outcome of this research has been the identification of specific notations and mechanisms for providing tolerance to software faults, including recovery blocks [3, 14, 18, 24] and N-version programming [7, 12]. Nevertheless, the utilisation of this approach in practical systems remains rather limited, although dual-software systems have been constructed for a number of critical systems [20, 21]. Again, the main reason for this may well be the lack of data on how effective this particular approach is in improving reliability. To date, the evaluation of software fault tolerance has either been performed by statistical modelling techniques [10, 13, 17, 23, 25] or by
empirical studies of multiple versions of software modules [8, 15]. Both of these modes of evaluation have their limitations. The modelling approach is often bedevilled by unjustified assumptions and/or unquantifiable parameters, whereas the empirical approach has usually had to be applied to relatively small modules (because of cost considerations). Nevertheless, both approaches have indicated the potential for significant gains in software reliability from the use of fault tolerance techniques.

This paper reports on a three year project (completed 1984) conducted at the University of Newcastle upon Tyne in conjunction with MARI, the Microelectronics Applications Research Institute. The aims of this project were:

1. To refine and develop software fault tolerance techniques for use in concurrent and real-time systems.

2. To confirm the utility of these techniques in a practical context.

3. To determine and quantify the effectiveness of the techniques for enhancing software reliability.

4. To measure the costs and overheads incurred as a consequence of adopting fault tolerance.

In order that the results of the project could be considered applicable and relevant to current practical systems it was decided to implement, for evaluation purposes, an application system of reasonable scale, constructed by professional programmers to normal commercial standards. The application selected was a medium-scale naval command and control system, engineered to be as realistic as possible, but incorporating software fault tolerance capabilities based on recovery blocks and "conversations" [14, 24].

An experimental programme was designed which involved executing the application software with a simulated tactical environment using a large number of action scenarios. Two modes of execution were available, depending on whether the fault tolerance features were enabled or disabled. Data from these experiments was analysed to provide a number of quantitative assessments of the improvement in reliability arising from the use of fault tolerance. In fact, the results of this analysis suggest that software fault tolerance can prove very effective in coping with the consequences of faults in software.

This paper provides an overview of the experimental configuration, describes the programme of experiments, summarises the data obtained from the experiments, and presents the analysis of and results derived from this data. Information on costs is briefly summarised in the conclusions. Project reports [2, 6] provide full details of the experimental configuration and programme, and also supply more details on costs. Further papers
are being prepared which describe the software fault tolerance techniques developed for this project [5] and recount the experience gained from their use [9].

2. Experimental System Configuration

The hardware configuration for the experimental system is illustrated in figure 1. Three DEC computers are employed to support the following sections of the system:

i) Command (PDP-11/45)

This machine supports the command and control system itself. The software is written in the CORAL language, and runs under a project developed MASCOT executive. (MASCOT is a design methodology for software construction and testing; the MASCOT executive is an operating system which supports and controls pseudo-concurrent processes and their interactions [22].) The command and control system was designed according to MASCOT techniques, and documented to the standard defined by JSP 188 (the U.K. Ministry of Defence standard for military systems); the involvement of the Royal Navy was sought to ensure that the system would be realistic in scale and functionality. The command and control system takes its input from simulated radar, sonar and inertial navigation systems, displays the information from these sensors on a Label Plan Display (LPD - a simulated radar display overlaid with track markers), and interacts with an operator, allowing him to conduct a "vectac" - a vectored attack on a hostile submarine by means of a helicopter armed with a torpedo. The command and control system consists of approximately 8000 lines of CORAL source code, structured into 14 concurrent activities as indicated in figure 2. The command and control system and its interfaces are summarised pictorially in figure 3.

ii) Simulator (LSI-11/23)

The software running on this machine (again written in CORAL and running under MASCOT) holds a data representation of the tactical environment, and simulates the sensors which provide the input to the command and control system (see figure 4). The details of the tactical scenario to be used for a run are read from a remote data file.

iii) File Server (PDP-11/45)

This machine provides file service facilities to the command and control system and to the simulator system. In particular, it is used for logging the monitoring data generated by the command and control system, and to hold details of the scenarios which drive the simulator.
In addition to the software mentioned above various items of support software (MASCOT run time executive, communications software, man-machine interface software, test and development software) were developed by the project.

Software fault tolerance was incorporated into the command and control software only, in the form of acceptance tests executed on completion of critical tasks, and alternate modules of independent design which could be executed in the event of an acceptance test detecting a problem. Automatic state restoration was available to attempt to eradicate errors from the system. These facilities were extended to provide recoverable "dialogues" between multiple processes [5]. A dialogue is an explicit embodiment of, and notation for, a restricted form of the concept of a conversation [24] which is, in turn, an extension to concurrent systems of the recovery block technique. As such, the dialogue construct could be used to impose restraints, appropriate to a MASCOT system, on the provision of recovery to concurrent activities, while still permitting inter-process communication.

The MASCOT operating system was modified and extended to provide recovery capabilities for processes and for information recorded in the shared data areas used for process interaction. These recovery mechanisms utilised a special purpose hardware device, called the recovery cache [19], which enables state restoration to be performed very quickly (the recovery cache may be thought of as providing a highly optimised implementation of checkpointing for multiple processes).

3. Experimental Programme

In order to measure the effectiveness of the software fault tolerance techniques in enhancing reliability a series of experimental runs were performed using various tactical scenarios to drive the simulator system. Three phases of experimentation were conducted. For each phase of experimentation the application software was frozen; that is, no changes were made to the command and control software during a phase of the experiments.

The first phase of experiments involved two versions of the command and control system. In version one the software fault tolerance was enabled and operated normally, whereas in version two, fault tolerance was disabled by the simple expedient of forcing all run-time checks to return a positive (ie ok) response.

It was originally intended that each experiment would consist of a pair of runs, one conducted with fault tolerance enabled, the other with fault tolerance switched off. The intention was that the unrecoverable run would proceed along a similar path to the fault tolerant run until an event occurred. The unrecoverable run would then provide data on the consequences of that event in a non-fault tolerant system. A test exerciser sub-system (automatic operator) was constructed to run in conjunction with the command
and control software in an attempt to provide a consistent operator reaction and thereby ensure repeatability. Experience soon showed, however, that the system did not provide the levels of repeatability required for such a method: two runs started in a similar manner were likely to follow quite different paths. The causes of this lack of repeatability are well understood, and centre around the unpredictability of the external interfaces to the command and control system. In particular, the communications protocol used to interface with the simulator (which uses a combination of checksums, timeouts and re-transmissions to ensure that no messages are lost or corrupted) is such that the ordering of the stream of messages from the simulated environment cannot be guaranteed to be the same for two runs conducted under similar conditions.

Because of these problems, fewer non-fault tolerant runs were performed, and only in the first experimental phase. However, the non-fault tolerant runs were used to provide information which enabled accurate predictions to be made of the effects of any event in the non-fault tolerant version. Furthermore, measurement of the reliability of the two versions enabled a direct confirmation to be obtained of the improvement in reliability due to fault tolerance. For phases two and three it was felt that our knowledge of the system was adequate to dispense with this confirmation, so all runs were performed with fault tolerance enabled.

In the second phase of experiments the same command and control software was used as for the first. In part, the intention was to confirm the results of phase one. More importantly, however, the first phase identified numerous problems with the MASCOT recovery software, and these were corrected for phase two. Since the success of the fault tolerance techniques is dependent on the recovery mechanisms, the results from phase two should more accurately reflect the benefits possible from fault tolerance in practice.

In the third phase of experiments, the command and control software was modified by replacing a number of modules with new versions written by inexperienced programmers. These versions were expected to contain a greater number and wider range of faults than the original modules. Furthermore, where original modules were retained, the sequencing of alternates in recovery blocks was reversed, so that the back-up alternates were used as primary alternates (and vice-versa). Any faults in the recovery mechanisms identified during phase two were rectified before phase three.

Two further phases of experimentation were envisaged, and one of these was attempted. The intention was to evaluate the effectiveness of the fault tolerance techniques at different levels of software reliability, and to this end, all faults identified in the application system during phase one were rectified to yield a more reliable version of the command and control software. Unfortunately, this system proved too reliable,
in that failure data was generated much too slowly. This phase of experimentation was therefore terminated unsuccessfully.

Time and financial limitations precluded the last phase of experimentation, in which it was planned to utilise an unreliable version of the application system derived from incompletely tested modules which had been archived during the development of the command and control software.

Each phase of experiments consisted of a number (60) of runs of the command and control system for which tactical scenarios were enacted on the simulator. Each run was monitored by the support system, and was carefully observed by an operator. Each time an event occurred (an event is either a system failure, or the detection of an error in the state of the system) the entire system would halt, and the operator would first log the incident and then analyse the error and attempt to identify the fault which caused it. The run would then be continued to see if fault tolerance would enable a failure to be averted, or if the failure would nevertheless occur. The system itself also recorded data to monitor events; the categorisation of events presented in the next section is based on these two sources of information. A run was considered to have finished when the scenario was completed, or when a failure occurred which prevented the system from continuing.

4. Experimental Programme Results

The results from the experimental programme are presented in two sections; the first section gives a summary of the events which occurred in the fault tolerant runs (for each of the three phases), whereas the second section presents overall statistics for the two versions of the system in phase one.

4.1 Summary of the Fault Tolerant Runs

The results in this section consist of a summary of the events which occurred during the fault tolerant runs of the experimental programme, where an event is defined to be either an observed failure, or the detection by internal checks of a suspected error in the state of the system.

In order to analyse the data from each run it was necessary to determine whether or not each event would have resulted in failure had the system contained no fault tolerance features. Usually the answers to such questions were obvious, but whenever there was any doubt surrounding the outcome of a particular event in a non-fault tolerant system, the option was available to run the system in non-fault tolerant mode and attempt to re-create the event in question. The effects of the event would then be directly observable. This was not found to be necessary in phases two and three of the experiments.
The following categories were used to group events according to their outcome:

**Events which yield an improvement in reliability over the non-fault tolerant system**

1. Events which produced recovery which averted failure.

**Events for which no change in reliability is produced in comparison with the non-fault tolerant system**

2. Events in which recovery occurred unnecessarily, but no failure resulted (usually a consequence of a faulty acceptance test).

3. Events in which a successful recovery took place, but the system failed nevertheless, as it would have done in the absence of fault tolerance.

4. Events in which recovery was attempted but was not successfully accomplished, and the system failed, as it would have done in the absence of fault tolerance.

5. Events in which the system failed without recovery being attempted, as it would have done in the absence of fault tolerance.

**Events which result in a deterioration in reliability when compared with the non-fault tolerant system**

6. Events in which a defective recovery caused the system to fail.

**Events for which the outcome is uncertain**

7. Events in which the effect on the system is unclear.

Two cases are considered; firstly a summary of all events, and secondly a summary of the first events which occurred in each run. This distinction is made to factor out any effects which might arise due to including events which occur after a non-fault tolerant system would have failed.
Total Number of Runs

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

**Summary of All Events**

1. Recovery averting failure 40 34 91
2. Unnecessary recovery 4 6 4
3. Recovery followed by failure 0 5 4
4. Defective recovery 13 18 17
5. Failure with no recovery 0 0 0
6. Failure caused by recovery 4 4 0
7. Outcome unclear 4 1 1

Total events: 65 68 117

**Summary of First Events**

1. Recovery averting failure 7 20 24
2. Unnecessary recovery 4 3 4
3. Recovery followed by failure 0 1 0
4. Defective recovery 9 10 5
5. Failure with no recovery 0 0 0
6. Failure caused by recovery 3 3 0
7. Outcome unclear 4 0 1

Total first events: 27 37 34

4.2 Comparative Data for Fault Tolerant and Non-Fault Tolerant Runs

Data in the previous section relates solely to runs of the fault tolerant version of the system, and assessment of the impact of fault tolerance on system reliability depends upon the analysis and categorisation of the events which took place. In this section, data is presented which enables a direct comparison to be made between the overall reliability of the two versions of the system. This may seem to provide a superior approach to comparative evaluation, but the reader is cautioned that due to a variety of factors (discussed below) the implications to be drawn
from this data must be stated less firmly than those based on the data of the preceding section. Because of this limitation, the data for this section was only collected during phase one.

The following table presents a summary of phase one of the experiments for both versions of the command and control system. It records the total number of experimental runs of each version, the total elapsed time during the execution of these runs, the total number of failures which occurred, and the number of runs which were completed without a failure of the command and control system (i.e. either completion of the scenario or premature termination due to a failure elsewhere).

<table>
<thead>
<tr>
<th></th>
<th>Fault Tolerant</th>
<th>Non-Fault Tolerant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total runs</td>
<td>43</td>
<td>17</td>
</tr>
<tr>
<td>Total run time</td>
<td>50423 sec.</td>
<td>28057 sec.</td>
</tr>
<tr>
<td>Total failures</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>Failure-free runs</td>
<td>24</td>
<td>8</td>
</tr>
</tbody>
</table>

A number of points must be taken into consideration when analysing this data:

1. Firstly, the nature of a run should be considered. A run begins with a period of relative inactivity, during which little other than object and screen updating and object classification takes place, and during which very few events occur. This is followed by a phase during which the system supplies the operator with information enabling him to guide an armed helicopter to engage a target submarine, referred to as a "vectac" (vector and attack), which constitutes a period of intense activity during which events are much more likely to occur. After the vectac the system returns to relative inactivity until either a further vectac takes place or the run is stopped. During the experimental programme, in order to restrict runs to a manageable duration, and to ensure an adequate rate of occurrence of events, the periods of inactivity before and after a vectac were artificially curtailed. This was done by running the simulation in "fast run" mode until shortly before the vectac was due to commence, then ending the run shortly after the vectac had completed (assuming that the system continued to run until this point). This curtailment has the effect that, since the system is likely to suffer few, if any, failures during periods of relative inactivity, any reliability measurements based on timing figures (for example MTBF) will appear far worse than they otherwise would. Thus, such figures might give a less favourable impression than figures for the proportion of events successfully recovered.
2. The lack of repeatability between runs (see section 3), the consequent lack of a one to one correspondence between fault tolerant and non-fault tolerant runs, and the divergence of the two systems when an event occurs, means that the implications of a direct comparison between the two systems are not as definitive as are the experimental results presented in the previous section.

3. To ensure that this data corresponded to all of the experimental runs in phase one, judgements were made concerning the four events in category 7 (outcome unclear). These events were classified as two failures (one each in categories 4 and 6) and two spurious recoveries. (The experimenters were reasonably confident that this classification was accurate, but the assignment was less certain than the categorisation given in section 4.1.)

4. The figures are heavily weighted by one particular run in which 16 of the total of 25 non-fault tolerant failures occurred. This run was in no way a 'freak'; all the failures were explained by known faults. However, the frequency of occurrence of such runs will clearly affect the overall system reliability. Unfortunately, there is insufficient data from the non-fault tolerant runs to deduce the probability that such a run will occur.

5. Analysis of Results

A number of different approaches can be adopted for estimating the increase in reliability which can be attributed to the provision of fault tolerance in the command and control system. Three approaches, characterising different aspects of reliability, are presented in the following sections. The first approach is based on estimating the "coverage" achieved by the fault tolerance techniques; that is, what proportion of potential failures are successfully averted thanks to software fault tolerance? The second approach provides a direct estimate of the mean time between failures for both versions of the system, while the third quantifies the proportion of missions successfully completed for the two versions of the system.

5.1 Coverage Analysis

The principal measure of the effectiveness of software fault tolerance was taken to be the "coverage" factor of these techniques; that is, the proportion of failures which would have occurred in a non-fault tolerant system that are successfully averted by means of fault tolerance. To be more precise, for situations in which the non-fault tolerant system would fail, coverage represents the probability that the fault tolerant system will nevertheless continue to operate without failing. The required probability can be easily estimated from the data of section 4.1 and thus relies solely on event counts. The coverage factor is calculated as the ratio of the number of failures averted (event category 1) to the number of potential failures
(event categories 1, 3, 4 and 5). Events in category 2 (spurious recovery) and category 7 (unclear events) are disregarded. Events in category 6 (failures introduced by fault tolerance) cannot be ignored, but are excluded from the initial calculation.

Thus, considering all events in phase one of the experiments the coverage achieved by fault tolerance is estimated to be 40/53, which is approximately 0.75. This is the maximum likelihood estimate. A Bayesian analysis using the Beta distribution indicates that the value estimated can be asserted to exceed 0.67 with 90% confidence. These figures should be abated to take into account the four failures caused by fault tolerance. The simplest approach regards these failures as "own goals" and subtracts them from the successes of category 1. An amended coverage estimate of 0.68 is then obtained.

The following table presents these coverage estimates for the three phases of experimentation. The estimates have been calculated for the two sets of data, namely all event data and first event data.

## Failure Coverage by Software Fault Tolerance

<table>
<thead>
<tr>
<th></th>
<th>All Events</th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Raw Coverage</td>
<td>0.75</td>
<td>0.60</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>Bayesian 90% point</td>
<td>0.67</td>
<td>0.52</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>Abated Coverage</td>
<td>0.68</td>
<td>0.53</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>First Events</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Raw Coverage</td>
<td>0.44</td>
<td>0.65</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td>Bayesian 90% point</td>
<td>0.29</td>
<td>0.53</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>Abated Coverage</td>
<td>0.25</td>
<td>0.55</td>
<td>0.83</td>
</tr>
</tbody>
</table>

### 5.2 Failure Rate Analysis

Simple arithmetic applied to the table of section 4.2 yields the following results:

- Failure rate for the fault tolerant system: 1.36 per hour
- Failure rate for the non-fault tolerant system: 3.21 per hour
- Ratio (fault tolerant / non-fault tolerant): 0.42

Making the standard, though often unjustified, assumption that the mean time between failures (MTBF) can be calculated as the reciprocal of the failure rate, yields the following alternative presentation of these results.
MTBF for fault tolerant system: 0.74 hours
MTBF for non-fault tolerant system: 0.31 hours
Ratio (fault tolerant / non-fault tolerant): 2.36

These results may be compared with those of the previous section by using the change in failure rate to provide an estimate for the coverage of failures by means of fault tolerance.

\[
\text{Failure coverage: } \frac{3.21 - 1.36}{3.21} = 0.58
\]

This value of 0.58 should be compared with the estimate of 0.68 (abated coverage, all events, phase one) presented in the previous section. The agreement is reasonably close, and the measurements are mutually supportive. However, it should be remembered (see section 4.2) that the comparison between the fault tolerant and non-fault tolerant runs is by no means exact, because of the inability to precisely repeat any individual run. In phases two and three, the failure rate for the fault tolerant system improved to 0.88 per hour and 0.58 per hour (MTBF 1.14 hours and 1.72 hours).

5.3 Successful Missions

A further comparison between the two versions of the system may be made by examining the proportion of runs which were completed without a failure arising from the command and control system. Again, from the table in section 4.2, it can be seen that the fault tolerant system is more reliable, though the improvement is much less marked.

<table>
<thead>
<tr>
<th>Description</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportion of fault tolerant runs which completed without failing:</td>
<td>56%</td>
</tr>
<tr>
<td>Proportion of non-fault tolerant runs which completed without failing:</td>
<td>47%</td>
</tr>
<tr>
<td>Ratio (fault tolerant / non-fault tolerant)</td>
<td>1.19</td>
</tr>
</tbody>
</table>

6. Discussion and Conclusion

The results of the previous section show clearly that for this application, in these experiments, the incorporation of software fault tolerance has yielded a substantial increase in reliability. Over the entire programme of experiments, the event counts show that 222 failures could have occurred due to "bugs" in the software of the command and control system. But of these 222 potential failures only 57 actually happened - the other 165
were masked by the use of software fault tolerance. This represents an overall success rate of 74%. (The same calculation restricted to first events yields the slightly lower figure of 67%.)

Examination of the results from the first phase of experiments suggested that much better results could be achieved if the underlying recovery mechanisms could be brought to an adequate standard of reliability. Essentially, the project was relying on prototype recovery mechanisms (the recovery cache and the MASCOT recovery software) to support the provision of fault tolerance at the application level. This situation would most certainly not be typical of an operational system where the recovery facilities should be at least as reliable as the hardware itself. It was hoped that improvement to the recovery routines for phase two would produce improved results, but in fact this effect was not observed until phase three. Projections suggest that with further improvements to the recovery software a coverage factor of over 90% could have been achieved.

The discrepancy between the results for all events and first events is very marked for phase one but is minimal in phases two and three. The most likely explanation is that the results for all events in phase one are rather better than they would otherwise be as a result of multiple recovery successes occurring in sequence. This phenomenon did occur in one spectacular case in phase one where a series of 12 successful recoveries in rapid succession helped boost the figures (and, to some extent, project morale).

Of course these gains were achieved at a cost, paid in capital costs to support fault tolerance, development costs to incorporate fault tolerance, and run time and storage costs to make use of fault tolerance.

The capital cost for supporting fault tolerance consisted of the cost of acquiring a hardware recovery device, for developing recovery software and incorporating this in the MASCOT operating system, and devising an interface by which dialogues and recovery blocks could interact with the operating system. The project expended approximately 1000 man-hours on these tasks, but the aim for the future would be that recovery facilities should be available on systems for critical applications on payment of a limited premium to the system manufacturer.

The supplementary development cost of incorporating fault tolerance in the command and control system was approximately 60%. This covered the provision of the acceptance tests and alternate modules used in recovery blocks and dialogues. The figure of 60% is probably rather high, reflecting the novelty of the techniques employed and their unoptimised utilisation in this particular application. Against the increased development cost must be offset any gains resulting in economies in testing the software.
Overheads in system operation were measured as: 33% extra code memory, 35% extra data memory and 40% additional run-time (though the system still had to meet its real-time constraints). The run-time overhead was incurred largely as a penalty for the synchronisation of processes for consistent recovery capability; data collection for state restoration purposes only contributed about 10% of the run-time overhead. By tuning the system to optimise its real-time response this overhead could be substantially reduced.

Our overall conclusion is that these experiments have shown that by means of software fault tolerance a significant and worthwhile improvement in reliability can be achieved at acceptable cost. We look forward to an independent confirmation of this result, preferably in the context of a system to be used in earnest.

Acknowledgements

Over the three year duration of this project we have received a great deal of assistance from many individuals, for which we are extremely grateful. We wish to express our gratitude explicitly to K. Heron, W. Lakin, B. Littlewood and B. Randell. The work was conducted with the financial support of the UK Science and Engineering Research Council and the Admiralty Surface Weapons Establishment of the MoD. We are grateful to these organisations and for the cooperation of the Royal Navy.
References


