CMU-CS-82-148

Specifying Reliability as a Software Attribute

Ivor Durham and Mary Shaw

Department of Computer Science,
Carnegie-Mellon University,
Pittsburgh, Pennsylvania.

6 December 1982

Abstract

This paper examines some issues in specifying reliability as a software attribute. A scheme for characterizing software reliability, known as a *failure profile*, is introduced. Failure profiles are derived for particular implementations of an abstraction by identifying analytically the behavior of the module when software or hardware faults occur. A failure profile is developed for a sorting program to demonstrate an informal technique for identifying the consequences of faults. The derived failure profile is compared with observations of the program's behavior in the presence of artificially induced faults to demonstrate the effectiveness of the failure profile characterization of software reliability. The issues raised in the application of the informal technique are discussed with respect to developing a formal and more mechanical technique for producing and using failure profiles.

This research was sponsored by the National Science Foundation under grant MCS-8011409.

Table of Contents

List of Figures

List of Tables

1. Introduction

The specification of a software module is intended to be a precise, but abstract, characterization of the module's behavior. Current software methodology emphasizes the importance of preparing a formal abstract specification for a module and limiting the information available to users of the module to precisely that specification. Typically a functional specification is prepared to guide the implementation effort, although in practice a specification may be revised in the light of specific implementation issues [18]. However, other software attributes in addition to functionality that would be useful in a specification are more difficult to prescribe before an implementation is attempted. Examples of such attributes are resource requirements (e.g., time and space) and reliability. Budgets may be prepared for these attributes based on anticipated use of the module, but these may be revised when actual use is observed. The budget may vary depending on the intended uses of the module. For example a string package used in an electronic mail system must not be allowed to fail in such a way that the destination of a message is switched to an alternate destination. However, occasional errors in the date or subject fields of the message would be acceptable to the mail system because these fields are not critical to the correct functioning of the mail transport mechanism. In this paper we address the problem of formally specifying software reliability. Specifically, we want to find a way of characterizing software reliability so that the reliability of different implementations of a particular abstract specification can be evaluated and compared. (Common measures such as Mean-Time-Between-Failures, MTBF, are too imprecise for design-level comparisons.) With tools to make such evaluations, a software designer can determine the effectiveness and the cost in time and space of different implementation that may contribute to improved reliability.

In order to show that an implementation meets its specification two things are required. First it must be verified that each function computes the correct result according to its specification, given the correctness of the virtual machine executing the function. Second, the assumption that the virtual machine is correct must be justified and discrepancies between the virtual machine and the actual computation must be accounted for. The technology required for verifying software has received considerable attention [3, 9]. The SIFT project, for example, has invested substantial effort in verifying the implementation of the whole hierarchy of system specifications from abstract requirements down to PASCAL code [19, 10] and corresponding benefits have been reaped in the discovery and correction of design flaws. However the verification ultimately depends on the correctness of the virtual machine provided by the PASCAL compiler and then on the underlying hardware. The assumption that the PASCAL compiler is correct is difficult to justify in the absence of its own verification. The correctness of the hardware is almost impossible to justify since it does not necessarily remain constant throughout its lifetime. The goal of this work is to address the second part of the problem: we want to make assertions about the behavior of the software when faults occur in the virtual machine executing a function, identifying how the faults are manifest if they are not processed and showing that the

effects of the faults are reduced when the faults are detected and handled in some fashion. These assertions may then contribute to a justification of assumptions that are based on the specification of the software.

In the SIFT project faults are manifested as discrepancies between outputs of identical, but independent, computations. However, massive redundancy may not be cost-effective or essential for tasks with different requirements. (For example, degraded performance may be acceptable for a limited time or high availability may be required at the cost of short periods of unavailability; neither situation is acceptable for the SIFT task of controlling an aircraft.) In many applications the manifestation of faults may not necessarily be treated uniformly. A technique is needed for making assertions about the behavior of software in the presence of faults. Formal techniques for making these assertions are important to the overall development of reliable software. However, no such formal technique is available at the moment. (The SIFT reliability analysis model characterizes the system state as a triple (h,d,f), where f represents the number of faults that have occurred, d represents the number of faults that have been detected, and h represented the number of fault that have been handled by reconfiguring the system. The analysis deals with the transitions from one state triple to another. While a simple state model is a promising basis for the analysis, the uniform characterization of faults in not suitable in more general cases.)

The problems involved in developing a formal reliability analysis technique were explored using an informal, but precise, analysis of one moderately complex program. Although the exploration was informal, it followed the same lines of analysis that formal techniques are expected to use. This approach should lead to formal techniques that will be applicable in practice and not confined to theoretical examples. Section 2 describes a means of characterizing the behavior of software in the presence of faults, known as a *failure profile*. The model on which failure profiles are based is described and the analytical derivation of profiles is discussed briefly. Section 4 describes a particular sorting program and a failure profile is developed for it in Sections 5 and 6. Specific predictions are derived from the failure profile in Section 7. These predictions were tested experimentally on the Cm* multiprocessor [5] by artificially increasing the fault rate so that the probability of a fault occurring during each sorting run was almost unity. The results of the experiment are presented in Section 8. Finally, issues raised in the use of the informal technique for deriving failure profiles are discussed in Section 9.

2. Failure Profiles

In trying to specify software *reliability*, the principal concern is actually to describe the ways the software can be *unreliable*. Software reliability may be characterized by a profile that describes the modes of failure that the software can exhibit as a consequence of faults. Section 2.1 describes a model, based on the principles of data abstraction, that focuses attention on individual components of a software system and allows the

effects of a fault to be considered in a limited context. The notion of a failure profile is defined in Section 2.2 and then Section 3 describes how profiles may be derived from the implementation of a particular abstraction.

In the sequel, we assume that a *fault* is an event that induces a change of state in a program that is neither intended nor desired by the program's designer. The new state is termed an *error state* if it remains within the formal specification of the program and a *failure state* if the new state violates the specification. (We usually omit the word "state" and refer only to errors and failures.)

2.1. The Model

A system is the composition of a hierarchy of abstractions, each of which has an abstract specification that describes its functionality and some other properties, including reliability (the goal of this work) and performance [15]. The construction of one abstraction must rely only on the specification, not the implementation, of other abstractions.

The detection of faults, errors, and failures is acknowledged across abstraction boundaries only through a well-defined, uniform exception mechanism [2, 8]. A fault and subsequent exception may be handled (in whole or in part) within any abstraction in the hierarchy that has sufficient context to understand the impact of the fault and to do something towards minimizing that impact. With an appropriate mechanism it might also be possible to correct the problem and resume processing at the point the problem was first detected [11, 8]. (For example, in a packet based communication network an exception may be raised when a packet's checksum is found to be incorrect. If the packet contains error-correcting information, the corrupted information may be restored and processing of the packet resumed.)

A fault that induces an error state should be detected and handled before the error state is used in normal processing, which may induce a second fault that may in turn result in a failure state. It is often an explicit action by the program that causes a fault (e.g. causing a memory access violation or attempting to read a non-existent file). However, the model permits the designer to trade off resources within the implementation of each abstraction to achieve the most cost-effective response to a problem. Saltzer demonstrated the principle with an example [12]: It may be more cost-effective to re-transmit a whole file over a network using knowledge of the file's contents to detect an error than to painstakingly protect the integrity of individual data packets used to transport the file.

Consider a typical network scenario in which a file transmission protocol is built upon several other layers of protocol including both message and packet transmission levels. The corruption of an individual packet may be handled at any of the levels: At the packet level, the lack of an acknowledgement may cause the

source of the garbled packet to re-transmit it; at the message level a lack of response or a negative response may cause the entire message to be re-transmitted; or at the file transmission level a similar response may result in the file transmission being restarted. Detecting and handling the fault at any of these levels is equivalent in that the file will ultimately be transferred correctly. However, the varying costs of detection and handling at each level may influence the particular choice. For example, common faults are usually handled within the packet or message levels because many other useful protocols make use of them. Handling common problems at the file transfer level (for example) would leave the implementors of other protocols to duplicate the error handling code for their particular protocols.

Also consider again the example introduced in Section 1 in which a string package is used in an electronic mail system. The integrity of the mail destination strings mat be considered sufficiently important to warrant a separate implementation of the string package. For some additional investment in space and time strings may be stored redundantly and then their integrity checked with each operation on them. Multiple implementations of one abstraction nay be more cost-effective than uniformly increasing the cost of all uses of the abstraction within the system.

2.2. Definition of a Failure Profile

For a particular implementation of an abstraction (data structure and set of functions), which we shall refer to as a *module*, we would like to be able to identify the set of failure states for the module and then we would like to be able to make assertions about the probability that each of the failure states may result when a particular function in the module is invoked. The collection of these failure states and their associated assertions then forms a *failure profile* for the module. The vulnerability of a module to a particular mode of failure depends not only on the implementation of the module, but also on the other modules used in that implementation. Hence the construction of a failure profile for one module depends on the failure profiles of the other modules it uses. Furthermore, changes in the failure profile of one module may alter the failure profiles all of the modules that use it. Individual failure profiles may be constructed for each function in a module and to call the collection of function failure profiles the failure profile for the module.

For the purpose of reliability analysis and specification, a number of distinct outcomes can be identified when a function of a module is invoked. For this particular investigation we recognize five general outcomes:

1. The function call fails to return.

2. It reports an inability to perform. (E.g., insufficient resources are available or a data structure fails a consistency test required for the function to perform correctly.)

3. It reports an incorrect result, but one that still satisfies the specification of the type. (E.g., $2 + 2 = 5$ for an integer result of the *Add* function.)

4. It reports an incorrect result that violates the functional specification of the type. (e.g., The function returns "code" 5 when only codes 1 to 4 are defined.)

5. It reports a correct result.

For each outcome, an assertion can be made about the likelihood of that outcome given that some fault occurs. The set of these assertions for a particular software function constitutes a *function failure profile*:

{P(*fail to return*), . . . , P(*report correct result*)}

In other words we want to be able to include as part of the specification of the function a statement of the form: "If a fault occurs within this function and that fault is either not detected or not handled by the current implementation, the probability that the function will not return at all is A%, that it will report an inability to perform is B%, ...; the probability of a correct outcome, notwithstanding faults within the function, is N%" and so on.

The kinds of failures and the level of detail included in the failure profile is tailored for each module. For example, a decision may be made to assign individual probabilities to the various circumstances in which a function can report an inability to perform. The designer is free to choose where in the hierarchy of abstractions it is appropriate to distinguish detected errors and where it is appropriate to group them and report the problem accordingly.

It is assumed that the failure profiles for primitive abstractions (e.g., primary memory or processor instructions) can be determined either analytically or empirically [16, 17]. This investigation is concerned with deriving a failure profile for a module from its implementation and from the reliability specification (failure profiles) of the more primitive modules it uses. The objective is to derive the overall failure profile for the program, given failure profiles for all abstractions in the hierarchy. The resulting program profile gives an estimate of the likelihood that a given computation will complete successfully in the presence of a fault.

For the multiprocessor program studied in this experiment, very simple primitive profiles were assumed.

Only one type of fault is admitted and that fault stops just one of the collection of concurrent processes that constitute the program. The fault does not destroy any data.

That is, all faults were assumed to result in a virtual machine function failing to return. This is equivalent to saying that the failure profiles for all of the primitive abstractions that are provided to the program by its execution environment (i.e the virtual machine) specify a constant probability of $1.0 - \epsilon$ for the outcome in which the function completes successfully, ϵ for the outcome in which the function call fails to return, and zero for all of the other outcomes. When a particular process is executing on the virtual machine and a virtual

machine function fails to complete, the process will also fail to complete its work. Hence the process and virtual machine failure profiles are identical in this particular case.

The effect of the strong restriction on the primitive failure profiles simplifies only the manual construction of failure profiles for the experimental program. The derivation of failure profiles, as is the case with most other attempts to analyze software, can require the manipulation of a substantial amount of information. Such manipulations are error prone when performed manually. The discovery of a formal technique for deriving failure profiles is likely to form a basis for automating this form of analysis. For the present the presentation is restricted to manually tractable examples.

3. Determining a Failure Profile for a Module

The purpose of a failure profile is to characterize the behavior of one specific module in the presence of faults. The definition of a failure profile for a module focuses attention on the individual functions exported by the module. There are two steps in deriving a failure profile. The first step is to identify the set of failure states that may be induced by a fault that occurs within the function or within the virtual machine executing the function. A failure state is also referred to as a "mode of failure." The second step in deriving a failure profile is to determine the relative frequencies of the modes of failure in the presence of faults.

The analysis needed to identify modes of failure is similar to that used to develop assertions for program verification; we do not have a formal technique to do this for modes of failure yet. All functions in the module are assumed to have been verified.[1] A formal technique for identifying modes of failure should be able to use the same assertions produced for the verification by relating modes of failure to clauses in the assertions becoming false.

For this preliminary investigation an informal technique for identifying modes of failure is used. A formal technique may be derived later from what is learned from applying this informal technique. An individual function is viewed as a state machine in which the states represent the aggregate of the information visible to the users of the module. Transitions between states are the consequences of sequences of program statements that change the externally visible information. Any assignment to an exported variable is considered to be a state change for now. Each non-control statement is the (possibly nested) application of a set of function calls. These functions are provided by more primitive abstractions for which failure profiles are already available.

For this investigation, the failure profiles for all primitive abstractions are assumed to be the same and to

[1] The verification was omitted for the experiment described in the following sections.

consist of only two outcomes: either the function completes successfully or it fails to return. The implementation of a particular function can be examined first to identify the statement sequences that lead to a change in externally visible state and second to identify the consequences of the sequence failing to complete because one function call failed to return. (In the experiment, a multiple process program is used so that a function failing to return does not necessarily mean that the program as a whole will fail.)

The modes of failure represented in the profile might be quite abstract in the sense of the five "generic" outcomes described in the previous section, or they may be much more finely distinguished. We do not know how to select an appropriate level of detail at this stage. It is important to attend to the problem of combinatorial growth across abstractions when deriving profiles for a hierarchy of modules. Profiles may be developed iteratively, introducing more detail in each iteration to help identify particular points of vulnerability that must be reduced to meet an acceptable level of fault-tolerance.

The relative frequency of each outcome in the profile when a function is invoked is proportional to the amount of time spent in the function executing statement sequences whose interruption by a fault would lead to the particular outcome. In most cases, except for faults that are completely transparent (e.g. faults that affect obsolete information), the outcome is in fact a mode of failure. Traditional complexity analysis is needed for this task [1, 6]. The functions that describe the relative frequency of each outcome of the function application constitute the failure profile for the function. The collection of individual function profiles constitutes the profile for the module.

In the following sections we describe a sorting program used on a multiprocessor and develop a failure profile for the sorting abstraction. The derived profile is then compared with experimental observations of the programs behavior under the same conditions that were assumed for the analysis of the program.

4. Program for the Experiment

To evaluate the effectiveness of the informal technique for deriving failure profiles, an empirical test was conducted. A program was designed and implemented, a failure profile was derived using the technique described above, and then the program's behavior was monitored as it was subjected to the type of fault anticipated by our assumptions. The general structure of the program chosen for this exercise is a set of identical, but independent, processes performing a simple, repetitive computation on discrete partitions of shared data. This structure is convenient because the program as a whole can survive the loss of some of its processes. Indeed, the loss of a processor was the failure mode selected for the experiment.

4.1. The Computational Task

The task for this experiment is to sort a modest number (up to a few thousand) of records. The sorting computation is a very simple quicksort, requiring only the comparison of pairs of keys and the exchange of pairs of records found to be out of order. The task was simplified by restricting records to a single data value that was also used as the key. More specifically, the sorting task permutes a sequence of 16-bit keys, $\langle k_1, k_2, \ldots, k_N \rangle$, so that in the resulting sequence, $\langle k_1', k_2', \ldots, k_N' \rangle$, $k_i' \leq k_{i+1}'$, for some definition of the "$\leq$" ordering. The Quicksort algorithm [3, 14] was used because it produces sets of sub-tasks that may be performed concurrently by different processes. For a given sequence of keys, $\langle k_1, k_2, \ldots, k_N \rangle$, the algorithm permutes the sequence around an arbitrary key, k_i, so that the keys $k_1', \ldots, k_{i-1}' \leq k_i' \leq k_{i+1}', \ldots, k_N$ ($k_i = k_i'$). The same partitioning algorithm is then applied to the subsequence that now precedes key "k_i" and to the subsequence that succeeds key "k_i". The sets of keys in these subsequences are mutually exclusive so that they may be manipulated by independent processes without synchronization. When the length of a subsequence produced by this splitting algorithm is less than some limit, M, a more direct sorting algorithm is used for the final ordering.

4.2. Program Design

The design of the program for a multiprocessor resembles quite closely that given by Knuth for a uniprocessor Quicksort program [7]. The special considerations for the multiprocessor environment and for this exercise are discussed below. They involve the way the data are shared and manipulated by the processes and the control structure under which the various processes cooperate to perform the task. The task of the Quicksort program is to sort a vector of several thousand numbers. For this experiment, the program generated its own data, which was either worst-case or random, depending on the particular experiment's parameters. (Worst-case data generates the maximum number of simultaneously pending sub-tasks.)

The organization of the data for the Quicksort program is shown in Figure 4-1. The data to be sorted are maintained as a simple vector in primary memory. All of the processes participating in the sorting task share access to the vector. A work-unit performed by a particular sorter is to split a sub-range of the key vector into two parts. A work-unit is described by a record that contains the indices of the first and last key vector entries in the sub-range to be split by the sorter. This is the same as the uniprocess implementation described by Knuth. The target multiprocessor, Cm* [5], provides a primitive *stack* data type that implements the usual stack discipline for a set of single-word values and enforces synchronized access to the stack so that multiple processes may attempt to push or pop the stack concurrently. One such *stack object* is used to keep pointers to pending work-unit records. The flow of work-units through the program is shown in Figure 4-1: The program is started by pushing the first work-unit onto the work stack (a) (that is the stack of pending work-units). One of the sorter processes removes the top work-unit from the work stack (b) and partitions it

using two new work-unit records obtained from the free stack (d). At least one of the new work-units is pushed onto the work stack (a) while the original work-unit record is returned to the free stack (c).

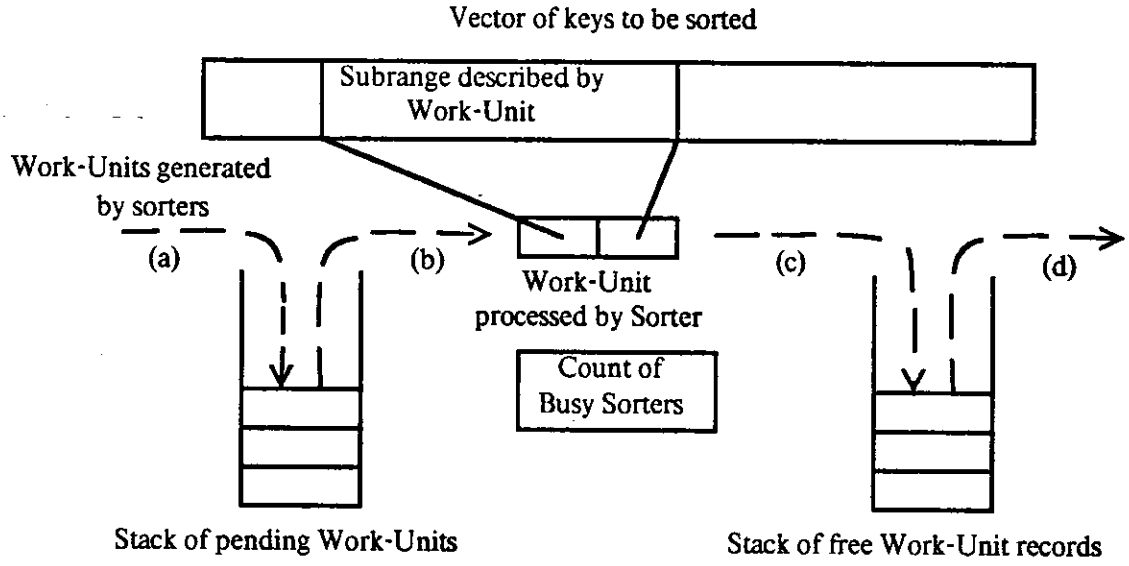

Figure 4-1: Program Data Structure

Because synchronization is applied to the system-supported stack, the program itself does not need to provide further synchronization while there is work to be done. However, the processes cannot assume that the sorting task is complete as soon as the work stack becomes empty; other processes may still be working towards making more work-units available. This problem is solved by having the sorters cooperatively maintain a count of the number of sorters actually processing work-units: When a sorter acquires its first work-unit after a period of being idle, it increments a *Busy-Sorters* counter that is shared by all of the sorters. Similarly, when a sorter finds the work stack empty, it decrements the *Busy-Sorters* counter.[2] When *Busy-Sorters* finally decrements to zero the entire task is complete.

Other than synchronizing access to the shared work and free stacks and to the *Busy-Sorters* counter, the sorters may execute autonomously. However, some initialization is required for the whole task: The first work-unit is initialized to (1,N), (representing keys $k_1, \ldots, k_N$) and pushed onto the work stack. The *Busy-Sorters* counter is initialized to the number of processes assigned to the sorting task—all sorters are busy initially. When the sorters see the count become non-zero, they begin their work and continue autonomously

[2] The Cm* multiprocessor provides indivisible increment and decrement operations to allow concurrent attempts to manipulate shared counters in this fashion.

thereafter. At the end of the sorting activity the user is notified that the program has terminated successfully. The initialization and completion tasks are performed by a *manager* process, which does not participate in the computation in any other way. (This control structure is *not* the more common master-slave organization in which a *master* process directs particular *slave* processes to perform specific units of work.) The manager and sorter organization is shown in Figure 4-2. The manager process is *not* included in the reliability analysis; its activity is limited to the very beginning and very end of the computation, the analysis of which is straightforward.

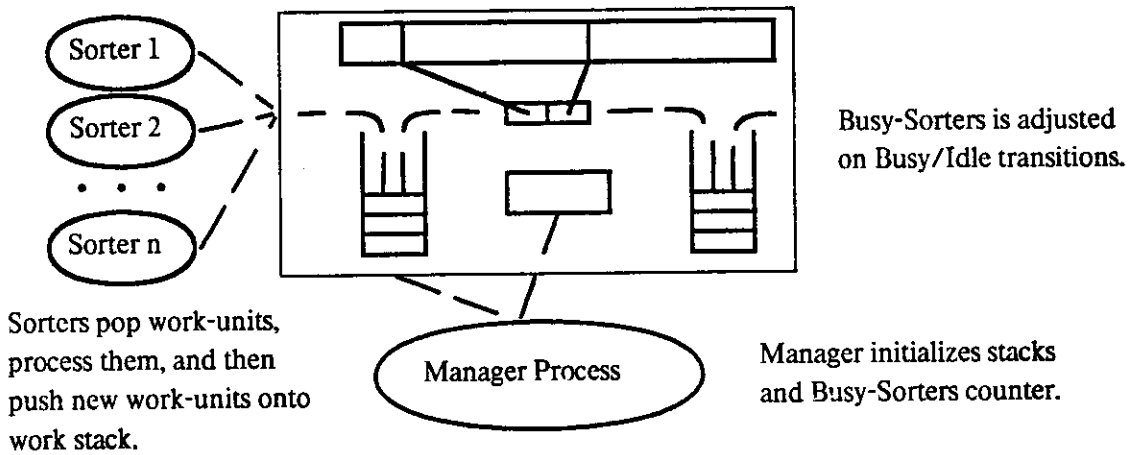

Figure 4-2: Program Control Structure

To avoid having the sorters consume processor resources while they were waiting for the manager to prepare the task, and to prevent the manager from consuming resources while it waits for the sorters to finish their work, we extended the logical semantics of the operations on the *Busy-Sorters* counter. When the manager changes the counter from zero to non-zero, it goes on to notify each sorter so that the sorter may begin work. Similarly when the count is decremented to zero by the sorters, the manager is notified (by each sorter that discovers the count is zero) so that it can inform the user that the task is complete. In other words, we made the zero/non-zero transitions of the counter observable events. This enabled us to suspend a process that was waiting for one such transition of the counter and have the operating system wake it up automatically.

4.3. Implementation of the Sorters

Figure 4-3 shows the structure of a sorter process and the actions it takes while working on the sorting task.

```
while Busy-Sorters > 0 do
 Pop work-unit off work stack
 if Pop succeeded
   then begin
          Increment Busy-Sorters if previously idle
          while true do
           Divide current work-unit into two new work-units
           if Both new work-units are empty
             then exit loop
             else Push larger of new work-units onto work stack if not empty and
                  keep smaller work-unit as current work-unit
           fi
          od
        end
   else Decrement Busy-Sorters if previously busy
 fi
od
```

Figure 4-3: Sorter Actions

The task of dividing the subrange described by the current work-unit into two new work-units is done by the *Divide* function. It fills in two new work-unit records for the left and right parts of the split subrange:

```
function Divide(Current,Left-Result,Right-Result: work-unit)
         returns (Left-Empty,Right-Empty: boolean)
```

If both of the boolean results are *true* then the sorter leaves the inner loop to acquire more work from the shared work stack. Otherwise the sorter makes one of the new work units available to the other sorters by pushing it onto the work stack. The sorter keeps the second new work-unit and divides it into two more work-units on the next iteration of the inner loop.

The Divide function uses a *Swap* function to permute pairs of keys (Figure 4-4). We observe that the Swap function is the only code to manipulate the key vector and therefore the only place at which the program itself could fail to maintain the permutation property of the sorting algorithm.

```
function Swap(Key1,Key2: key)
 begin
  Temp: key;
  Temp := Key1;
  Key1 := Key2;  This statement destroys the permutation property of the key vector
  Key1 := Temp   This statement restores the permutation property of the key vector
 end
```

Figure 4-4: Swap Function

When a work-unit describes a sufficiently small subrange of the key vector, a different sorting algorithm is used to avoid the overhead caused by manipulating the work stack and the extra calls on the Divide function. We chose the Bubblesort algorithm [7] for this program because it relies only on the *Swap* function to manipulate the data.

5. Program Statement Sequences and Modes of Failure

A constant failure profile is assumed for the virtual machine that executes the sorters. This assumption results immediately in an identical profile for the individual sorters. (If the virtual machine function fails to return, the sorter does not execute any more and so fails to return to its invoker.) Because the program involves multiple sorters and shared data, the changes in shared state caused by the sorters need to be examined (even though the failure profile for the sorter itself is available). In this section the statement sequences within a sorter are identified and the corresponding modes of failure derived.

5.1. Sorter Activity

The computational activity of a sorter may be described as a set of statement sequences, each of which leads to some externally visible state change. The sorter activity from Figure 4-3 is expanded in Figure 5-1 to show the statement sequences in more detail. The manager process initializes the shared variable `Busy-Sorters` is initialized to 2^p.

```
Statement {The private variable Busy-Sorters is initially true}
Seq. No.
1234567

123 567 while Busy-Sorters > 0 do
123 567   Pop work-unit off work stack
123 567   if Popped succeeded
            then begin
  3 56            if not Busy
                    then begin
  3                       Increment(Busy-Sorters);
  3                       Busy := true
                          end
                  fi;

    4 6           while true do
    4 6             Divide current work-unit into two new work-units
    4 6             if Both new work-units are empty
    4 6               then exit loop
                      else Push larger of new work units if not empty and
    4 6                    keep smaller work-unit as current work-unit
                    fi
                  od
                end
          else begin
12      7         if Busy
                    then begin
1       7               Decrement(Busy-Sorters);
1       7               Busy := false
                        end
                  fi
                end
        fi
      od
```

Figure 5-1: Statement Sequences Executed by Sorters

In the following description of the statement sequences, a superscripted asterisk "*" indicates that the par-

ticular sequence is repeated an arbitrary number of times and a superscripted hash-mark "$^{\#}$" indicates that the sequence is repeated at most once. Each statement sequence is assigned a number which is printed in Figure 5-1 to the left of each statement in the sequence. The functionality of each sequence is described below:

1. All sorters, except the first, find the work stack empty initially. This will cause them to change their state from busy to idle (`Busy := false`). The program statements that effect this change of state are referred to collectively as the "Become-Idle" sequence:

   Become-Idle ::= {*Test-Busy-Sorters* (>0); *Pop-Work-Stack* (fail);
   Test-Busy (true); *Decrement-Busy-Sorters*}$^{\#}$

   All sorters also execute this sequence once at the end of the computation and thereby decrement *Busy-Sorters* to zero.

2. When a sorter is already idle and it continues to find no work it executes the "Still-Idle" sequence:

   Still-Idle ::= {*Test-Busy-Sorters* (>0); *Pop-Work-Stack* (fail); *Test-Busy* (false)}*

3. When a sorter acquires work for the first time after being idle it executes the "Become-Busy" sequence:

   Become-Busy ::= {*Test-Busy-Sorters* (>0); *Pop-Work-Stack* (succeed);
   Test-Busy (false); *Increment-Busy-Sorters*}$^{\#}$

4. Once a sorter has acquired its first work unit from the stack it must process it and perhaps push a new work unit onto the stack. This is referred to as the Do-Work sequence:

   Do-Work ::= {{*Divide*; *Push-One-Unit-Work-Other*}*; {*Divide*; *Both-Results-Empty*}}$^{\#}$

5. Once the sorter is officially Busy it repeatedly acquires work units from the stack and processes them. The "Still-Busy" sequence is responsible for acquiring work:

   Still-Busy ::= {*Test-Busy-Sorters* (>0); *Pop-Work-Stack* (succeed); *Test-Busy* (true)}

6. The sorter executes the *Still-Busy* and *Do-Work* sequences repeatedly:

   Keep-Busy ::= {*Still-Busy*; *Do-Work*}*

   This sequence is repeated until the work runs out at which point the sorter becomes idle.

7. When there are no more work units the sorters execute the *Become-Idle* sequence. Then they execute *Still-Idle* until *Busy-Sorters* becomes zero:

   Finish-Up ::= {*Become-Idle*; *Still-Idle**}

There is some overlap in the definition of the statement sequences; some sequences are defined in terms of

others. However, this is necessary for accumulating the times spent in the sequences that result in external state changes. When a particular statement belongs to more than one statement sequence each particular execution of the statement must be charged to the appropriate statement sequence.

5.2. Modes of Failure

The assumed failure profile for all primitive abstractions expects that the only failure mode is that a function fails to return. Using this assumption the statement sequences executed by a sorter can be examined to determine how the program as a whole might fail when there is a failure in an arbitrary primitive abstraction (*i.e.* when one of the sorters is stopped by a "lightning bolt" and therefore fails to complete a particular sequence of statements). The effects of failing to complete the statement sequences are described below. In some cases it is not possible to predict whether the keys will ultimately be permuted correctly. In such cases a pessimistic, but safe, prediction is made on the assumption that the keys cannot be assumed to be correctly permuted in the presence of a fault.

Sequence	Mode of failure
Become-Idle	One sorter fails to inform the others that it has become idle. It does not have a work unit in hand so the program as a whole *fails to terminate, but the final permutation of the keys is correct anyway.* (The other sorters handle all of the work remaining after the one sorter dies.)
Still-Idle	Once a sorter has become idle, its death due to a lightning bolt is completely transparent to the other sorters. (It has no work-unit in hand and it is already counted as idle in *Busy-Sorters.*)
Become-Busy	As a sorter acquires its first work unit after being idle it must inform the others that it has become Busy. If the sorter has acquired a work unit but dies before incrementing *Busy-Sorters*, the program may terminate prematurely with an incorrect result: The dead sorter has not sorted its work unit, but as far as the other sorters are concerned it is still idle, therefore they can decide to terminate. (Note: This is where the notion of sequences rather than individual statements becomes important. In the Become-Idle and Still-Idle sequences sorters continue to execute Test-Busy-Sorters and Pop-Work-Stack, but these execution times are charged to the sequence of which they are a part. As Test-Busy-Sorters and Pop-Work-Stack are both part of two sequences the time the program spends on these statements depends on the time the program spends in the various statement sequences containing them.)
Do-Work	The sorter is actively processing a work unit so it is counted as busy and the program as a whole *fails to terminate and the keys are not permuted correctly.* Of course, its data might coincidentally be in order already. Therefore it is possible that a correct result still may be computed even though the program fails to terminate; again, the safe assumption is made.

Still-Busy The sorter does not have a work-unit in hand until it completes *Pop-Work-Stack* successfully. The sorter is counted as busy, so the program as a whole *fails to terminate and the keys are not permuted correctly.* This is the sequence only when the *Pop-Work-Stack* operation will succeed; otherwise the program is executing the *Become-Idle* sequence. If, however, the fault occurs before the Pop instruction is executed, the program as a whole may go on to compute the correct result because the process is not in possession of a work-unit when lightning strikes.[3]

Keep-Busy See *Still-Busy* and *Do-Work*, above.

Finish-Up See *Become-Idle* and *Still-Idle*, above.

In summary, there are four distinct outcomes that may occur when a lightning bolt stops a sorter:

1. The program fails to terminate, and the keys are not permuted correctly into the final order (i.e. no result at all is delivered). This mode of failure is referred to as "Not Terminate, Not Correct". It arises from faults in sequences *Do-Work* and *Still-Busy*.

2. The program terminates, but the result is incorrect: "Terminate, Not Correct". This failure mode arises from faults in the *Become-Busy* sequence.

3. The program fails to terminate, but the final permutation of the keys is correct anyway: "Not Terminate, Correct". This failure mode arises from faults in the *Become-Idle* sequence.

4. Finally, the loss of the sorter is completely invisible and the program terminates with the data permuted correctly: "Terminate, Correct". This successful termination mode arises from faults in the *Still-Idle* sequence—and, of course, the absence of faults.

6. Relative Frequency of Failure Modes

In this section a simple complexity analysis is done to determine how much time is spent in each statement sequence. To simplify the analysis two sets of assumptions are made. The first set concerns the environment in which the program executes and the second set concerns specific properties of the data. These assumptions simplify only the task at hand and do not affect the ultimate effectiveness of the approach being explored.

[3]This distinction was not made in the analysis, but the experimental results indicate that this distinction is important and might alter the distribution of outcomes in the analysis. This may suggest that the particular choice of sequences was not appropriate.

6.1. Environmental Assumptions

- Only one kind of fault is admitted, the effect of which is to halt a process permanently [13]. This fault is referred to as a *lightning bolt.* No data are damaged by this fault.

- Lightning bolts are independent of program activity and in any given set of program runs the distribution in time of the arrival of lightning bolts is uniform. (I.e. the program may be struck at any time during its execution with equal probability.)

- Only sorter processes may be struck by lightning and the distribution of lightning bolts across the set of sorters is uniform. The manager is immune from lightning strikes.

- The inter-arrival time of lightning bolts is more than the run time of the program for any particular set of data. This ensures that there is at most one fault to be considered while the program is executing.

- The probability of failure of primitive system operations is zero. This covers the operations on the work and free stacks as well as the manipulation of the shared *Busy-Sorters* counter, all of which require system-provided synchronization.

6.2. Analytical Assumptions

To compute actual probabilities some assumptions are made about the data because they affect the length of time that a particular sorter is busy:

- Each subrange is assumed to be exactly bisected by the Divide function.[4] The program uses a variant of the Divide function that picks the middle key as the fence between the two subranges [7]. The data to achieve this experimentally consists of a set of distinct keys 1..N, N is odd, where the original sequence is $N, 2, N-2, 4, N-4, 6, \ldots, 3, N-1, 1$.

- The number of sorters working on the task is 2^p, for some p.

- The number of records to be sorted supports the bisecting assumption [7]: $N = 2^k(M+2) - 1$, for some k, where M is the minimum sequence length before the Bubblesort is used instead of Quicksort partitioning.

- The records are all distinct. (This prevents a failure in the Swap function from resulting in a sorted set of keys, but one that is not a permutation of the original sequence.)

- All sorters are involved in the activity at some stage; that is, $k \geq p$.

---

[4] It was initially assumed that the subrange given to Bubblesort was in reverse order, but this was difficult to guarantee in the experiment.

- The actual running times for individual statements can be measured experimentally.

6.3. Analysis

It is clear that the probability of the program failing is the same as the probability that one sorter will be struck by a lightning bolt that leads to one of the three fatal modes of failure. To derive the probability of each mode of program failure is the fraction of the program's execution time that is spent in statement sequences that are vulnerable to that specific failure mode, given that a lightning bolt arrives. This is done by examining the statement sequences and estimating how many times each is executed by a sorter. The times that a sorter spends in the sequences that contribute to a particular failure mode are summed to give the total time the sorter is vulnerable to that failure. The arguments used to derive this information are typical of complexity analyses. It is not the goal of this work to formalize such analyses; a formal technique would be useful, however [15].

Based on the assumptions that the number of keys to be sorted is $N = 2^k(M + 2) - 1$ and the number of sorters is 2^p, the overall behavior can be described as follows: Initially only one sorter has any work to do and the remaining sorters become idle. When the original sequence of keys has been split into two sub-ranges by the first sorter, a second sorter can go to work on one sub-range, while the first sorter continues to work with the other sub-range. As the two sorters complete their work, each producing two more sub-ranges to be split, two more sorters can go to work, so a total of four sorters are now active. Each sorter in turn produces two more sub-ranges, pushing one onto the shared stack and keeping the other to work on. This progression continues until all of the sorters are active.

Assuming total equality and perfect parallelism among the sorters, this activity can be viewed as a sequence of stages in which each active sorter is working on the same sized work-unit. In moving from one stage to the next, the number of pending work-units is doubled as is the number of sorters that can be working on the task. As soon as each sorter has a work-unit of its own, the analysis can proceed as if for the rest of the computation it works solely on sorting the sub-range given in that first work-unit. That is, it repeatedly divides its first "private" subrange into two parts and then divide those parts in turn until the original subrange is completely sorted. This ignores the mechanics of putting one of the two work-units produced at each stage onto the shared work stack and, indeed, the actual computations—but the analysis depends only on the sizes of the work-units and their distinction. However, the assumption that all sorters are working with perfect parallelism allows us to make this simplification for the purposes of analysis.

The computational activity of any particular sorter can then be described in two parts. First the sorter generates work for other, currently idle, sorters. After all sorters are busy, all of the sorters do exactly the

same amount of work on subranges of equal size. The next two sections analyze in detail the number of times each statement sequence is executed in firing up other sorters and in working on the "private" subrange.

6.3.1. Start-up activity

The amount of work done by a sorter before all of the sorters are busy may be expressed in terms of the sorter's logical index (1..2^p), although in practice which sorter gets to a particular work-unit first is non-deterministic. The first sorter executes *Still-Busy* once and some number of {*Divide*; *Push-One-Work-Other*} sequences while providing work for other sorters. All other sorters execute {*Become-Idle*; *Still-Idle**} until work becomes available. When work finally becomes available, some sorter P_i, i > 1, executes the *Become-Busy* sequence once and some number of {*Divide*; *Push-One-Work-Other*} sequences until all sorters are busy. Without loss of generality the sorters can be numbered so that the number of the latter sequences for sorter P_i is $p - \lceil \log_2 i \rceil$ i.e. P_1 executes it $p - 0 = p$ times, P_2 executes it $p - 1$ times, P_3 and P_4 execute it $p - 2$ times, and so forth, before all sorters are busy.

6.3.2. Common activity

The number of stages in a computation is $k + 2$. This comes from the assumption that the number of keys is $N = 2^k(M + 2) - 1$ and the assumption that the current subrange is always bisected, which doubles the number of work-units at each stage and approximately halves the size of those work-units.[5] Hence the number of stages during which *all* sorters are active is $h = (k + 2) - p$ and the work done by each sorter once all of the sorters are busy is:

- {*Divide*; *Push-One-Work-Other*} for each subrange that is not Bubblesorted. This is a binary progression for which the sum is $2^{h-1} - 1$.

- {*Divide*; *Both-Results-Empty*} for each subrange that is Bubblesorted. This is 2^{h-1} times.

- *Still-Busy* whenever a new work-unit is taken from the stack (This is the same as the number of times the sorter reduces a subrange to the Bubblesort size, M): $2^{h-1} - 1$ times.

- *Become-Idle* once when the stack is empty. Given the assumption that work-units are bisected, once busy a sorter will either produce two new work units, one of which is pushed onto the stack and the other is used for the next iteration, or it will completely sort its current work unit and have to pop the stack to obtain its next work unit. The assumption that all sorters proceed at the same speed guarantees that any attempt to pop the stack will succeed once all the sorters have become busy. Hence, no sorter can execute *Become-Idle* twice after first acquiring work-units.

---

[5] The new size $S' = (S-1)/2$ since the pivot key is left in its final position and two equal sized sub-ranges are left on either side of that key.

Still-Idle cannot be counted effectively, but it doesn't contribute to program failure, so it can safely be ignored.

7. Program Failure Profile

A failure profile for the Quicksort program is a set of probability functions:

$$\{P_{p,k}(\neg Terminate, \neg Correct), P_{p,k}(Terminate, \neg Correct),$$
$$P_{p,k}(\neg Terminate, Correct), P_{p,k}(Terminate, Correct)\}$$

The probability of overall survival is computed by subtracting the three failure probabilities from 1.0:

$$P_{p,k}(Terminate, Correct) =$$
$$1.0 - P_{p,k}(Terminate, \neg Correct) - P_{p,k}(\neg Terminate, Correct) - P_{p,k}(\neg Terminate, \neg Correct)$$

The detailed profile depends on the configuration of the program as parameterized by p and k which determine the number of sorters working on the task (2^p) and the amount of data to be sorted ($N = 2^k(M + 2) - 1$). Each probability is computed from:

$$P_{p,k}(Mode) = \frac{P(Lightning\ Strike) \times (Time\ program\ vulnerable\ to\ mode\ of\ failure)}{2^p \times (Total\ program\ execution\ time)}$$

In Sections 7.1, 7.2, and 7.3, the preceding analysis is used to determine the amount of time that the program is vulnerable to each mode of failure. Section 7.4 shows the total program execution time. Section 7.5 provides numerical values of the failure profile for a variety of program configurations. For this experiment $P(Lightning\ Strike)$ is artificially increased to 1.0 with a distribution such that exactly one fault will occur while the program is running. The results are conditional probabilities of the various failure modes given that a fault has occurred. The data reported should therefore be interpreted as giving relative distributions of the failure modes for lightning-bolt induced failures, not absolute failure rates.

7.1. Time vulnerable to "Not Terminate, Not Correct" failure

The program is vulnerable to a failure in which the program neither terminates nor produces a correct result whenever the process incurring the fault is actively processing a work-unit. The statement sequences involved are *Still-Busy*, *Push-One-Work-Other*, *Both-Results-Empty*, and *Divide*. The Divide, Bubblesort, and Swap functions together require a fairly complex and data-dependent complexity analysis. To avoid this in the current investigation, the time taken by the Divide function was measured for subsequences of the sizes obtained by repeatedly bisecting the original sequence. This gives an enumerated function that can be used in the predictions (see Section 7.5). Hence the time that sorter P_i is vulnerable to this mode of failure is given by the following equation. To simplify the notation, let the name of the statement sequence stand for its execution time:

$$Process\text{-}Busy(i) =$$

$$Still\text{-}Busy \times (2^{h-1} - 1 + (\text{if } i = 1 \text{ then } 1 \text{ else } 0)) +$$
$$Push\text{-}One\text{-}Work\text{-}Other \times ((2^{h-1} - 1) + (p - \lceil \log_2 i \rceil)) +$$
$$Both\text{-}Results\text{-}Empty \times 2^{h-1} +$$
$$\sum_{j=\lceil \log_2 i \rceil + 1}^{p} Divide(PartitionSize(j))) + \sum_{j=p+1}^{p+h} (2^{j-p-1} \times Divide(PartitionSize(j))).$$

where $PartitionSize(x) = (N + 1) \times 2^{1-x} - 1$, the size of a work-unit at level x in the tree (counting from one). The last line of the equation represents the time taken in bisecting work-units. The first term represents the number of times the sorter provides work for an idle sorter. The second term represents the work done on the "private" work-unit after all sorters are busy. (Recall that $h = (k + 2) - p$ so the upper limit of the second term is the same as $k + 2$.) The total time that the program is vulnerable to this mode of failure is then:

$$\sum_{i=1}^{2^p} Process\text{-}Busy(i)$$

7.2. Time vulnerable to "Terminate, Not Correct" failure

As a process makes the transition from idle to busy, it leaves the program as a whole vulnerable to premature termination between the time a fresh work-unit is popped from the stack and the time the shared *Busy-Sorters* counter is incremented. The duration of this vulnerability is exactly the time taken by the Become-Busy statement sequence. Hence for a single process the time is:

Become-Busy

The total time that the Quicksort program is vulnerable to this mode of failure during a particular run is:

$$\sum_{i=2}^{2^p} Become\text{-}Busy$$

The first process never makes the transition from idle to busy because by the assumptions it is busy initially and never becomes idle.

7.3. Time vulnerable to "Not Terminate, Correct" failure

The program will compute a correct result, but fail to terminate, if a process is killed as it is changing state from busy to idle and thereby fails to decrement the shared *Busy-Sorters* counter. All sorters execute the *Become-Idle* sequence once at the end of the computation. In addition, all sorters except the first execute the sequence once at the beginning as they become idle waiting for some work to do. Hence the time that sorter P_i is vulnerable to this mode of failure is:

$$Become\text{-}Idle \times (\text{if } i = 1 \text{ then } 1 \text{ else } 2)$$

Hence, the total time that the program is vulnerable to this mode of failure is:

$$(2^{p+1} - 1) \times Become\text{-}Idle$$

7.4. Total Program Execution Time

The total running time for the program is the time that the first sorter, P_1, is active. This is simply the sum of the times that sorter P_1 is vulnerable to any of the modes of failure. (Remember that P_1 starts out busy and remains busy until *all* of the work has been done. We are analyzing the special case in which all slaves may be assumed to terminate at the same time having done the same amount of work in the common sub-tree. Thus *Still-Idle* need not be considered.) Hence:

$$Total = (Still\text{-}Busy \times 2^{h-1}) + Push\text{-}One\text{-}Work\text{-}Other \times (2^{h-1} - 1 + p) + \sum_{j=1}^{p} Divide(PartitionSize(j))) + \sum_{j=p+1}^{p+h} (2^{j-p-1} \times Divide(PartitionSize(j))) + Become\text{-}Idle.$$

7.5. Computed Failure Profile

The times taken to execute specific statement sequences were measured (see Table 7-1) and used to predict profiles for a set of configurations of the Quicksort program given that a specific fault occurs. Figures 7-1, 7-2, 7-3, and 7-4 show the trends in each component of the profile for data configurations $k = 1$ to 9 as the number of processors is varied, $p = 0$ to 5. Remember that $k \geq p$ so curves terminate when $k < p$. Samples from each of these Figures are shown in Tables 7-2, 7-3, and 7-4. In Section 8, these samples are compared with numbers collected experimentally.

The probability of $P_{p,k}(\neg Terminate, \neg Correct)$ rises to a maximum between $k = 7$ and $k = 8$. While this behavior seems counter-intuitive, it is attributable to the summation of terms in which each term is the product of two functions, one increasing, the other decreasing. The time to execute the Divide function was measured only up to $k = 9$.

Become-Busy	620	Become-Idle	613	Still-Busy	354
Push-One-...	1582	Both-Empty	340		
Divide(9215)	924890	Divide(575)	29904	Divide(35)	3495
Divide(4607)	275164	Divide(287)	15771	Divide(17)	3039
Divide(2303)	119298	Divide(143)	8642	Divide(8)	8725
Divide(1151)	58317	Divide(71)	5083		

Table 7-1: Time in milliseconds to execute statement sequences.

K	Keys	Not Terminate Not Correct	Terminate Not Correct	Not Terminate Correct	Terminate Correct
4	287	96.2	0.124	0.369	3.3
7	2303	97.2	0.014	0.042	2.8
8	4607	96.9	0.007	0.021	3.1
9	9215	95.2	0.003	0.01	4.8

Table 7-2: Failure profiles (%) for $p = 1$ when $P(fault) = 1.0$.

K	Keys	Not Terminate Not Correct	Terminate Not Correct	Not Terminate Correct	Terminate Correct
4	287	86.3	0.335	0.773	12.6
7	2303	89.6	0.040	0.091	10.2
8	4607	89.0	0.019	0.044	10.9
9	9215	84.6	0.009	0.020	15.4

Table 7-3: Failure profiles (%) for $p = 2$ when $P(fault) = 1.0$.

K	Keys	Not Terminate Not Correct	Terminate Not Correct	Not Terminate Correct	Terminate Correct
4	287	68.8	0.124	1.320	29.3
7	4607	74.9	0.037	0.079	25.0
8	2303	75.9	0.078	0.016	23.9
9	9215	67.9	0.016	0.034	32.0

Table 7-4: Failure profiles (%) for $p = 3$ when $P(fault) = 1.0$.

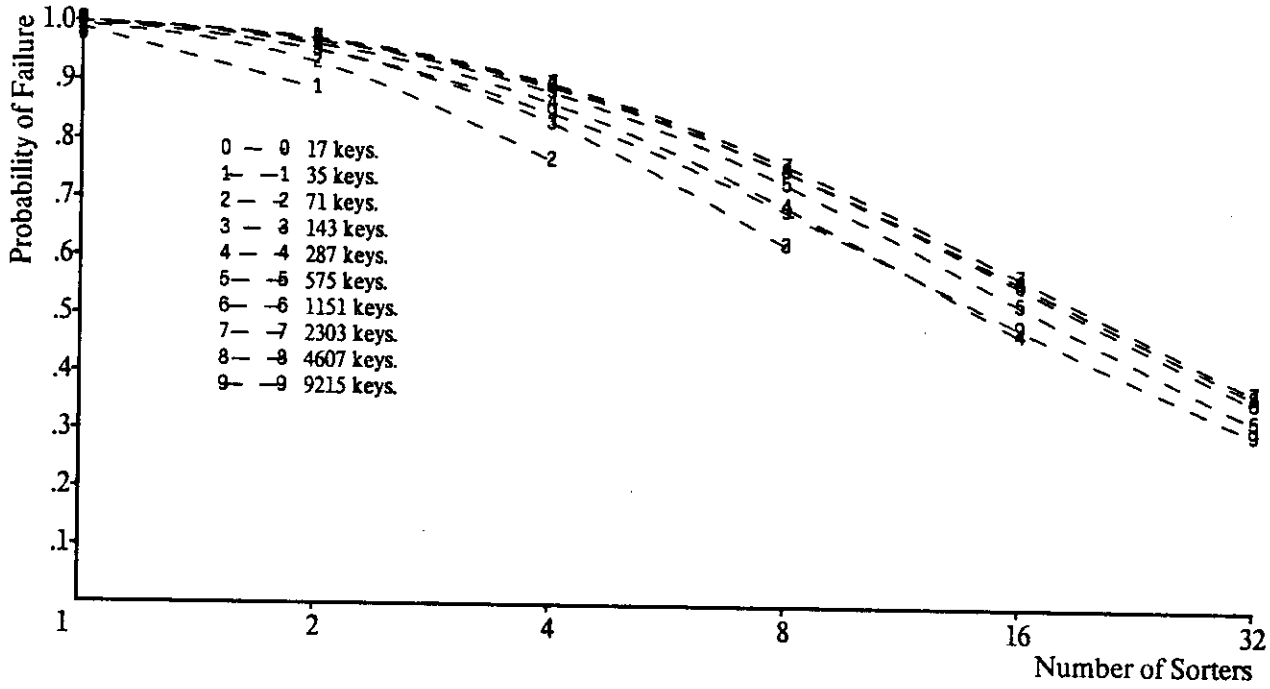

Figure 7-1: Probability of "Not Terminate, Result Incorrect" outcome.

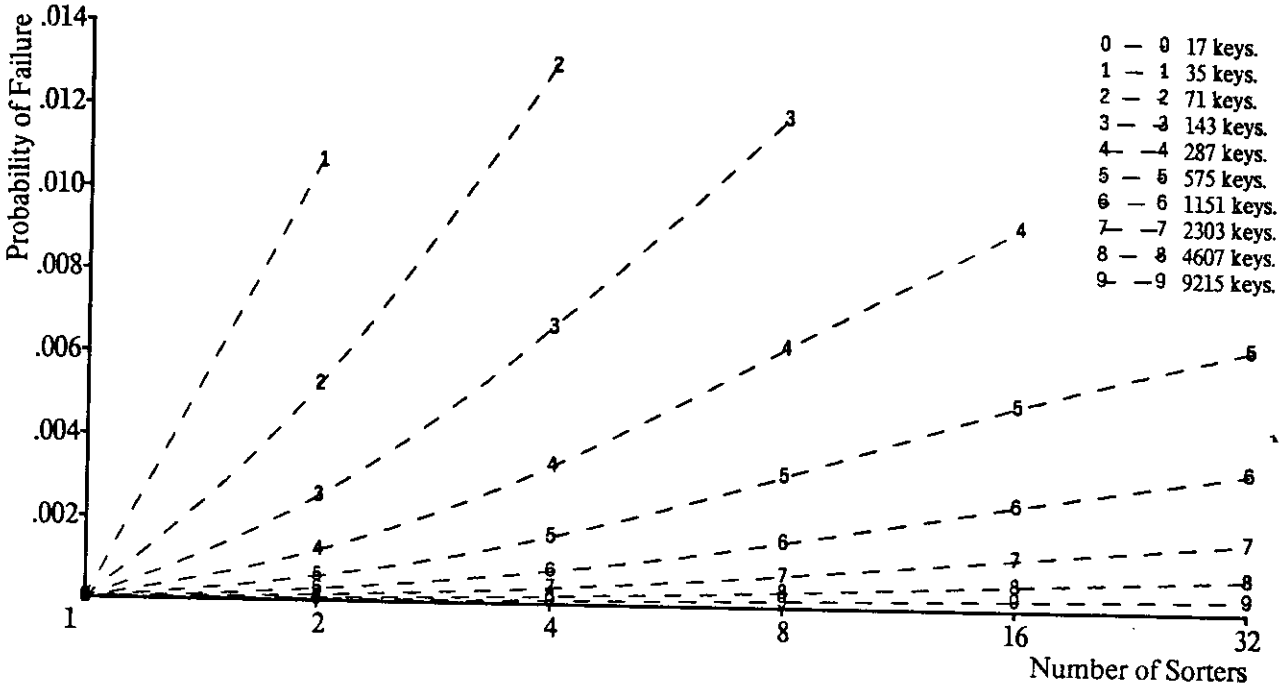

Figure 7-2: Probability of "Terminate, Result Incorrect" outcome.

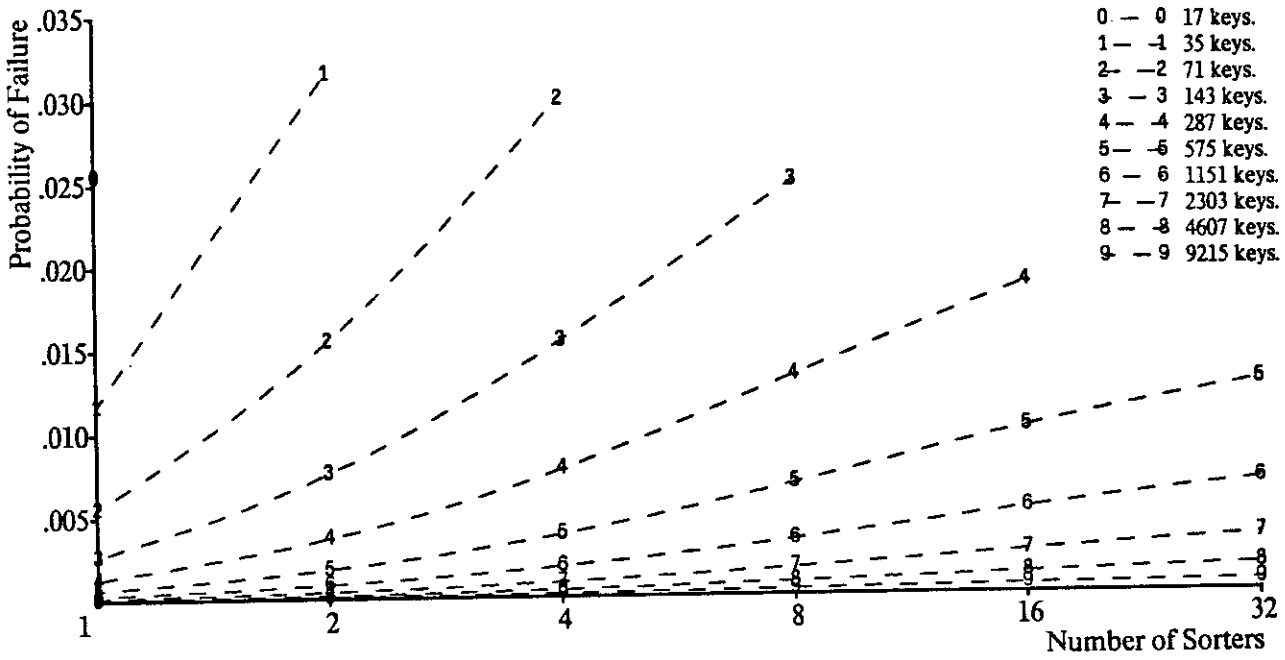

Figure 7-3: Probability of "Not Terminate, Result Correct" outcome.

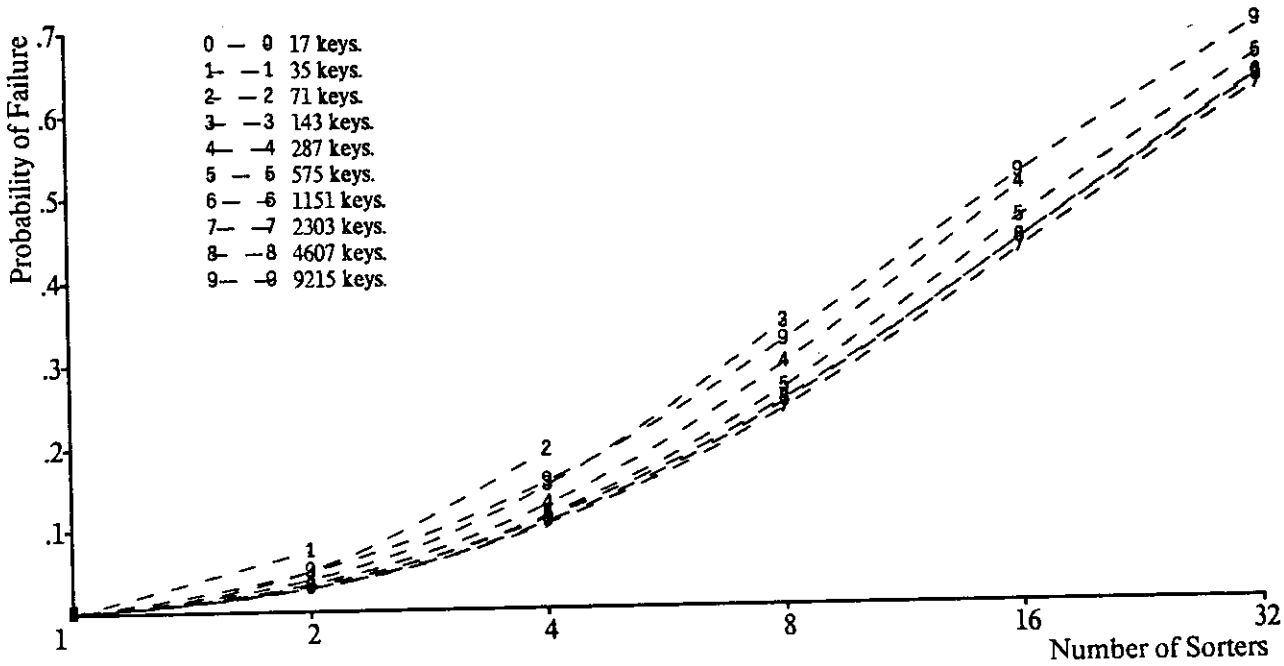

Figure 7-4: Probability of "Terminate, Result Correct" outcome.

8. Validation

To determine the accuracy of the predictions made from the analysis of the Quicksort program, some experiments were conducted on the Cm* multiprocessor [5]. The principal concern is the accuracy with which the effect of a given fault can be predicted. The Quicksort program was executed normally, but the frequency of occurrence of the fault allowed by the model, a "lightning bolt" that kills a process, was artificially increased to the point at which the probability of a lightning strike during a given Quicksort run was very close to 1.0. Section 8.1 describes the experimental design. Data corresponding to the predictions in Tables 7-2, 7-3, and 7-4 were obtained by running the Quicksort program a number of times and recording its behavior *only* when a fault occurred during the run. These results are presented and discussed in section 8.2.

8.1. Experiment Design

Of the five general outcomes identified in Section 2.2 only three are possible with this experiment:

- The program fails to return.
- The program returns a correct result.
- The program returns an incorrect result.

However, two different modes of failure can be distinguished when the program fails to return:

- The program cannot complete the computation (when the sorter struck by lightning is actively doing work).
- The program completes the computation correctly, but fails to terminate (when the sorter has no work in hand, but has not yet been counted as idle).[6]

To collect the information required to distinguish these outcomes, two processes were added to the set that constitutes the basic Quicksort program. The first additional process controls the experiment. It takes as input a series of experiment descriptions, specifying the configuration for the Quicksort program (number of keys to be sorted, number of sorters), the number of repetitions of the experiment for this configuration, and so on. For each experiment, it behaves as the "user" of the Quicksort program, invoking the program and waiting for it to return. If the program fails to return, the control process times it out. Independent of whether the program returns, the control process checks the vector of keys to see whether it is sorted. Finally, it records the results of the individual experiments for subsequent analysis.

---

[6] This would allow recovery to be made in some instances by timing out the program, ensuring that it had quiesced, and then checking its results. This approach might be more cost-effective than repeating the whole computation.

Faults are caused by the second additional process, the *Lightning Bolt*, which operates independently of the rest of the processes. The Lightning Bolt is invoked by the control process just before the Quicksort manager process is given the parameters for the current sort task. The control process tells the Lightning Bolt the expected running time of the Quicksort program so that the delays before the Lightning Bolt induces a fault are uniformly distributed across that running time.

The Lightning Bolt actually kills a Cm* processor rather than a sorter process. Since the failure profile of a sorter process was defined to match the failure profile of the processor, this has the correct effect. When a Cm* processor is killed forcibly, the StarOS Operating System [4] automatically recovers the processor and restarts it. StarOS notifies the process that was running on the processor that lightning struck. The process sees this as a spontaneous exception (like an interrupt).[7] The only modification made to the original Quicksort program for this experiment was to alter the sorter's default exception handler to send a detailed report of the fault-induced exception back to the control process. The report from the exception handler indicates the time of the fault, the sorter's state (program counter, value of the "Busy" variable, the size of the work-unit that was being processed by the sorter, etc.) All such reports from the individual runs of a particular experiment are recorded by the control process for subsequent analysis. Having sent off the report of the exception, the sorter returns to its starting point to await another invocation of the Quicksort program. This simulates a "permanently" halted processor that magically becomes available for a subsequent experiment.

8.2. Experimental Results

Experiments were run for each of the configurations for which predictions are tabulated in Section 7.5 (Tables 7-2, 7-3, and 7-4). The results of these experiments are presented in Table 8-1. For each particular combination of k (keys) and p (sorters), the Quicksort program was run 150 times. Since the fault rate was artificially increased to study the behavior of the Quicksort program in the presence of a fault, only those runs that were struck by lightning are tabulated; those runs that were completely missed by the lightning bolt were discarded. The number of runs that were actually hit by the synthetic lightning bolt are shown in the table.[8]

For each mode of failure, the table shows the number of times that each failure mode occurred (#), the percentage of all hits represented by that failure mode (%), and the percentage estimated by the failure profile in Section 7.5 (est).

---

[7] Only processors executing sorters were struck by the Lightning Bolt.

[8] The Lightning Bolt process was tuned to strike as many runs as possible with a uniform time distribution. It was difficult to constrain the process to strike during the expected running time of Quicksort, so in the runs that were not hit, Quicksort terminated before lightning struck.

			Not Terminate Not Correct			Terminate Not Correct			Not Terminate Correct			Terminate Correct		
k	p	Hits	#	%	est.	#	%	est.	#	%	est.	#	%	est.
4	2	131	120	92.3	96.2	0	.000	.124	11	.085	.369	0	0.0	3.3
4	4	136	106	77.9	86.3	0	.000	.335	28	20.6	.773	2	1.5	12.6
4	8	12	6	50.0	68.8	0	.000	.124	3	25.0	1.32	3	25.0	29.3[9]
7	2	143	124	86.7	97.2	0	.000	.014	17	11.9	.042	2	1.4	2.8
7	4	112	95	84.8	89.6	0	.000	.040	14	12.5	.091	3	2.7	10.2
7	8	88	67	76.1	75.9	0	.000	.078	8	9.1	.016	13	14.8	23.9
8	2	149	115	77.2	96.9	0	.000	.007	24	16.1	.021	10	6.7	3.1
8	4	112	99	88.4	89.0	0	.000	.019	10	8.9	.044	3	2.6	10.9
8	8	72	50	64.4	74.9	0	.000	.037	6	8.3	.079	16	22.2	25.0
9	2	149	117	78.5	95.2	0	.000	.003	19	12.8	.010	13	8.7	4.8
9	4	140	104	74.3	84.6	0	.000	.009	13	9.2	.020	23	16.4	15.4
9	8	79	45	57.0	67.9	0	.000	.016	7	8.9	.034	27	34.2	32.0

Table 8-1: Preliminary Results

There were no recorded instances of "Terminate, Not Correct." However, the number of occurrences of "Not Terminate, Correct" is uniformly higher than predicted. A detailed examination of the instances that failed in this way revealed that they were almost all inside one of the detailed sorting functions: Swap, BubbleSort, or Divide. This suggests that the test data was sorted prematurely so that a lightning strike had no effect on the ultimate correctness of the program's result. In some cases, the actual sorting activity on the work-unit had been completed, but the administrative task of freeing the work-unit record had not been completed.

The predicted failure profile is uniformly conservative, predicting failure in more cases than occurred in practice. Several decisions that may contribute to this have already been noted. If profiles are derived for the primitive abstractions, it is possible that those profiles may admit some probability of survival in the presence of a fault. This is in contrast to the assumption that the primitive profiles allow only for total failure.

Assuming that the data were not sorted prematurely and then counting the times premature sorting occurred under "Not Terminate, Not Correct" as they would be in the very worst case, the results are closer to the predictions made by the failure profiles. Table 8-2 shows the modified results with the "predicted percentage" columns replaced by the differences between the experimental and predicted percentages. The results are within about 10% of the predictions made from the analytical failure profile.

Actual data collected on the Cm* multiprocessor [16] show that any particular computer module (Cm*

[9] Eight processors working on only 287 keys complete the computation quickly, leaving little time for a lightning bolt to strike.

			Not Terminate Not Correct			Terminate Not Correct			Not Terminate Correct			Terminate Correct		
k	p	Hits	#	%	dif.	#	%	dif.	#	%	dif.	#	%	dif.
4	2	131	131	100.	+3.8	0	.000	-.124	0	0.0	-.369	0	0.0	-3.3
4	4	136	132	97.0	+10.7	0	.000	-.335	2	1.5	+.728	2	1.5	-11.1
4	8	12	9	75.0	-6.2	0	.000	-.124	0	0.0	-1.32	3	25.0	-4.3
7	2	143	140	97.9	+.7	0	.000	-.014	1	0.69	-.648	2	1.4	-1.4
7	4	112	108	96.4	+6.8	0	.000	-.040	1	0.89	-.02	3	2.7	-7.5
7	8	88	75	85.2	+9.3	0	.000	-.078	0	0.0	-.016	13	14.8	-9.1
8	2	149	139	93.3	-3.6	0	.000	-.007	0	0.0	-.021	10	6.7	+3.6
8	4	112	109	97.3	+8.3	0	.000	-.019	0	0.0	-.044	3	2.6	-8.3
8	8	72	56	77.8	+2.9	0	.000	-.037	0	0.0	-.079	16	22.2	-2.8
9	2	149	134	89.9	-5.3	0	.000	-.003	2	1.3	+1.29	13	8.7	+3.9
9	4	140	117	83.6	-1.0	0	.000	-.009	0	0.0	-.020	23	16.4	+1.0
9	8	79	52	65.8	-2.1	0	.000	-.016	0	0.0	-.034	27	34.2	+2.0

Table 8-2: Results

processor) could expect a transient error about every 130 hours.[10] The StarOS operating system does not attempt to mask these errors and the processor taking the error effectively commits suicide and waits for the automatic reconfiguration mechanism to restart it. This is exactly the model used for the experiment, although the failure rate was accelerated substantially. The Quicksort run times measured in the experiment in the presence of a fault (including a timeout in the control process) never exceeded 25 seconds for any configuration reported above. For a configuration with eight sorters we can deduce that the expected probability of a failure during a Quicksort run is in practice:

Faults per second per Cm: $F_1 = 1/(130\times3600)$

Faults per second in 8 independent Cms: $F_8 = 8 \times F_1$

P(*Fault in Quicksort*) $= 25 \times F_8 = 4.274 \times 10^{-4}$

Hence, an inaccuracy in the predictions of about 10% is in reality a change in the probability of an overall outcome on the order of 10^{-5}.

The differences between the predicted and experimental rates at which the program completely survives a fault was attributed to a lack of complete uniformity in the distribution of times at which lightning struck the program. This is a difficulty with the experimental environment used. However, conducting the experiment in a practical environment is more convincing than results from some form of simulation system.

[10] More recent data increases this interval to about 500 hours.

9. Issues

Failure profiles characterize in a quantifiable manner the anticipated behavior of a module in the presence of faults. However, this experimental investigation has highlighted some obvious difficulties with the simple techniques used to derive the profiles. These issues are discussed briefly below.

The first problem concerns how states and state transitions are defined in terms of the externally visible state of a module and sequences of statement within functions of the module. In the example program two components of the program state were identified, one of which affects the correctness of the program's results and the other affects the termination condition of the program. The operations performed on either part of the state were quite simple and it was straightforward to identify the statement sequences involved. It was assumed in the analysis that the program could not produce the correct result when the sorter that was struck by lightning is actively processing a work-unit. (That is the profile for the virtual machine was used instead of deriving a separate one.) However, the experimental data showed that the distinction between correct and incorrect results when the program fails to terminate could have been more accurate if more detailed profiles had been used.

In examining the statement sequences to identify potential modes of failure, one must be careful to ensure that there is only one mode of failure that would result from the interruption of the sequence. In the analysis of the *Still-Busy* sequence we did not distinguish between a fault before and a fault after the instruction to pop a work-unit off the stack. While the program would fail to terminate in either case, it would still compute a correct result if the Pop were not attempted since the work-unit would be left on the stack for another sorter to process. This appears to have contributed to some errors in the predictions. Hence, the sequence should have been split just before the Pop instruction.

The oversight in identifying multiple potential modes of failure within one sequence raises the issue of how to be sure that *all* of the potential modes of failure have been identified. In the degenerate case one would have to examine each individual program statement and decide upon the potential mode of failure if a fault were to occur between consecutive statements. In this case a formal technique could help both to limit the granularity at which the program examination is needed and it might provide the foundation for an automatic mechanism to relieve the burden on the designer or programmer.

The task of identifying potential modes of failure from the statement sequences is complex. In the worst case, a different mode of failure may result depending on which individual instruction is being executed when the fault occurs. In the example program there were only two relationships of importance that could be affected by a fault (the count of busy processes and the ordering of the sort keys). In the general case there may be many relationships of importance, but the complexity of the task might be reduced by focussing on statement sequences that affect individual relationships.

In the experiment there were only a few distinct program behaviors in the presence of the single type of fault that allowed by the assumptions. In more complicated programs a larger space of possible behaviors can be expected. The problem is then to decide how those behaviors should be represented in the profile: whether to provide a function for each mode of failure or whether to group some modes together.

The decisions made in defining the components of the failure profile influence the next problem, which concerns how the failure profiles for one module can be used to derive the profiles for higher level abstractions using the module. This problem was simplified in the example by having a two-element profile for all lower-level abstractions. Recall, also, that although there was a simple profile for an individual sorter process (function), the internal behavior of the sorter still had to be examined to derive the profile for the program as a whole.

The analysis that is necessary to determine the proportion of the total execution time that a function is vulnerable to a particular mode of failure is very similar to that used for complexity theory. While strict formal analyses could be quite expensive, an informal approximation method such as that used here might be more desirable because it is quite inexpensive to perform and may indeed be considered a "throw-away" as a particular design evolves. The benefit of a formal technique might once again lie in providing a basis for automated support for the designer.

Finally, it was suggested in the introduction that a designer might want to use the information provided by failure profiles to compare various implementations of a particular abstraction. The goal of such comparisons is to determine the cost-effectiveness of particular measures to reduce the vulnerability of the software to the faults identified by the failure profile analysis. A formal comparison will require both the failure profile analysis and an analysis of the time and space (and development?) costs of the various implementations. Evaluating such comparisons require either a policy statement against which to judge the improvements or some form of linear programming exercise to select the most appropriate implementation from those available or proposed.

There is some potential for manual or automatic program transformations for which the effects on failure and cost profiles are known. For example, knowing that the collection of slaves may not agree to terminate if one dies during the sort, the manager might eventually time-out and perform a quick check on the results, reporting success if all of the keys are properly ordered. (This is exactly what the monitoring process did to collect the experimental data.) The cost of making the check can be computed; the probability that the program fails to terminate, but computes a correct result may then be substantially reduced or even eliminated.

10. Conclusions

An extension to the specification of a module has been proposed. This extension captures the anticipated behavior of the module in the presence of specific, anticipated faults that are not explicitly detected or handled within the module. The functional specification of an abstraction may be complemented with a *Failure Profile* for each of the various implementations of the abstraction. The designer needing the abstraction as a component of a software system may then select one of the implementations according to the faults its operating environment may generate and the available budget for providing fault-tolerant coverage of those faults.

The intent of this investigation was to identify the difficulties involved in developing a formal technique for deriving failure profiles that capture accurately the behavior of a program in the presence of specific faults. It is apparent that most of the difficulties that were described in the previous section could be relieved to a considerable extent by a formal technique.

The straightforward analysis proved to be reasonably accurate in practice. The principal differences between the predictions and the experimental results were due to the granularity at which we examined the statement sequences, treating each logical statement as an single event. However, the analysis never predicted success inaccurately, so more success in the presence of faults is both beneficial and welcome. The method used to derive the failure profiles did not require a complete, formal analysis of the program to identify the major modes of failure [2].

We have demonstrated that the ideas presented in this paper may be used in practice. More important, we believe that they constitute a good basis for a more formal investigation that will eventually lead to more complete and rigorous, though still practical techniques, for identifying and evaluating potential modes of failure of software.

Acknowledgements

We are grateful to Anita Jones and Jon Bentley both for technical discussions and for their encouragement.

References

1. A.V. Aho, J.E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2. Flaviu Cristian. Exception Handling and Software-Fault Tolerance. Proceedings of the 10th International Symposium on Fault Tolerant Computing, IEEE, Kyoto, Japan, 1980, pp. 97-103.

3. C. A. R. Hoare. "Quicksort." *Computer Journal 5* (April 1972), 10-15.

4. A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl. StarOS, A Multiprocessor Operating System for the Support of Task Forces. Proceedings of the Seventh Symposium on Operating Systems Principles, ACM/SIGOPS, Pacific Grove, California, December, 1979, pp. 117-127.

5. Anita K. Jones, Edward F. Gehringer (eds.). The Cm* Multiprocessor Project: A Research Review. Tech. Rept. CMU-CS-80-131, Carnegie-Mellon University, Department of Computer Science, July, 1980.

6. Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1973.

7. Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

8. Roy Levin. *Program Structures for Exceptional Condition Handling.* Ph.D. Th., Carnegie-Mellon University, 1977.

9. Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

10. P. Michael Melliar-Smith and Richard L. Schwartz. "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System." *IEEE Transactions on Computers C-31*, 7 (July 1982), 616. 630

11. Brian Randell. "System Structure for Software Fault Tolerance." *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 220-232.

12. Saltzer, J.H., Reed, D.P., and Clark, D.D. End-to-End Arguments in System Design. Proceedings of the 2nd International Conference on Distributed Computing Systems, Paris, April, 1981, pp. 509-512.

13. Richard D. Schlichting and Fred B. Schneider. An Approach to Designing Fault-Tolerant Computing Systems. Tech. Rept. TR 81-479, Cornell University, Department of Computer Science, November, 1981.

14. R. Sedgewick. *Quicksort.* Ph.D. Th., Stanford Computer Science Department, May 1975.

15. Mary Shaw. A Formal System for Specifying and Verifying Program Performance. Tech. Rept. CMU-CS-79-129, Carnegie-Mellon University, June, 1979.

16. D. P. Siewiorek and Vittal Kini (eds.). Reliability in Multiprocessor Systems: A Case Study of C.mmp, Cm*, and C.vmp. Tech. Rept. CMU-CS-78-143, Carnegie-Mellon University, Department of Computer Science, 1978.

17. D.P. Siewiorek and R. Swarz. *The Theory of Reliable Systems Design.* Digital Press, Bedford, MA, 1981.

18. William Swartout and Robert Balzer. "On the Inevitable Intertwining of Specification and Implementation." *Communications of the ACM 25*, 7 (July 1982).

19. John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P.M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control." *Proceedings of the IEEE 66*, 10 (October 1978), 1240. 1255