AD-A 228, 180

NASA CR-182,106

NASA Contractor Report 182101
ICASE Report No. 90–60

NASA-CR-182101
19910002117

ICASE

PARALLELIZED RELIABILITY ESTIMATION OF RECONFIGURABLE COMPUTER NETWORKS

David Nicol
Subhendu Das
Dan Palumbo

Contract No. NAS1–18605
September 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

FOR REFERENCE

NOT TO BE TAKEN FROM THIS ROOM

LIBRARY COPY

NOV 0 1 1990

LANGLEY RESEARCH CENTER
LIBRARY NASA
HAMPTON, VIRGINIA

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

NF00903

Parallelized Reliability Estimation of Reconfigurable Computer Networks

David Nicol *
Subhendu Das
College of William and Mary

Dan Palumbo
NASA Langley Research Center

Abstract

This paper describes a parallelized system, ASSURE, for computing the reliability of embedded avionics flight control systems which are able to reconfigure themselves in the event of failure. ASSURE accepts a grammar that describes a reliability semi-Markov state-space. From this it creates a parallel program that simultaneously generates and analyzes the state-space, placing upper and lower bounds on the probability of system failure. ASSURE is implemented on a 32-node Intel iPSC/860, and has achieved high processor efficiencies on real problems. Through a combination of improved algorithms, exploitation of parallelism, and use of an advanced microprocessor architecture, ASSURE has reduced the execution time on substantial problems by a factor of one thousand over previous workstation implementations. Furthermore, ASSURE's parallel execution rate on the iPSC/860 is an order of magnitude faster than its serial execution rate on a Cray-2 supercomputer. While dynamic load balancing is necessary for ASSURE's good performance, it is needed only infrequently; the particular method of load balancing used does not substantially affect performance.

*This research was supported in part by the Army Avionics Research and Development Activity through NASA grant NAG-1-787, in part by NASA grant NAG-1-1132, in part by NASA grant NAS-1-18605, and in part by NSF Grant ASC 8819373.

N91 - 11430#

1 Introduction

For some time reliability analyses of fault-tolerant flight control systems have used automated tools such as ARIES, SURF, CARE III, ASSIST, and SURE for determining system failure probabilities[7] (also see the excellent survey in [8]). On large reliability models these programs require a great deal of computational effort. Typically the complexity of model analysis grows exponentially in the size of the model, forcing design engineers to use relatively simple reliability models. These tools simply are not equal to the challenge posed by the analysis of highly complex, highly reliable control systems such as the Integrated Airframe Propulsion System Architecture (IAPSA) [3]. These tools stand to benefit from parallel processing, if parallelism can be found and efficiently exploited.

This paper concerns two tools, ASSIST and SURE, that are used to place upper and lower bounds on the probability of failure in computer systems which can reconfigure themselves in the event of failure. We describe how these tools were rewritten to clearly expose parallelism, were extended to permit the construction of highly complex reliability models, and were subsequently ported to a distributed memory parallel architecture, the Intel iPSC/860. The parallel run-time performance on 32 nodes of the iPSC/860 is one thousand times faster than the run-time of the original tools on a Sun 3/150. This dramatic gain is brought about by a combination of algorithmic improvements, parallelism, and use of the advanced (in 1990) Intel 80860 [4] microprocessor architecture. We exploit the parallelism inherent in searching state-spaces, a topic of active research interest (e.g, see [6, 18, 16]). Our contribution is to show how to transform a reliability model into a form that can be efficiently and automatically processed on a parallel architecture. Our experience empirically proves the potential of parallel processing on a class of applications which hitherto have not exploited parallelism. Our performance comparisons between the iPSC/860 and Cray-2 also empirically prove the clear superiority of parallel processing on scalar applications of this type.

Our parallelized tool dynamically balances the workload whenever some processor goes idle. On the largest problems studied dynamic load balancing was called so infrequently that its cost does not detract greatly from the overall performance. However, failure to use dynamic load balancing can significantly degrade performance.

This paper is organized as follows. §2 provides background information on reliability modeling, and the tools of interest. §3 introduces the ASSIST language, §4 describes the model analysis underlying the reliability tools, and §5 describes how we combined and parallelized two existing tools. §6 reports on the measured performance of our tool on a suite of network reliability problems. §7 summaries this paper.

2 Background

We are interested in numerically estimating the reliability of complex highly reliable computer systems. The notion of a "state-space" pervades reliability analysis. Every component of a system is said to have a state. A reliability modeler is free to permit a component to

have any one of a set of states; typical component states are GOOD, BAD, IN_USE, UNUSED. Components may fail; failures may trigger recovery processes such as reconfiguring around a failed component or replacing it. The time-to-failure distributions for individual components are taken to be exponential; the time-for-recovery distributions are permitted to be general. Formally, the reliability model is a semi-Markov process [17].

A semi-Markov state-space can be thought of as a directed graph, each node of which is a system state. A system state is typically described as a vector of integer-valued component state values; the system state changes when some component state changes, for example, if a processor or communication link fails. The amount of time the system spends in a state is random, and is known as the *holding time*. Arcs out of a state describe *transitions* that are possible from that state. The behavior of the modeled system over a time interval $[0, T]$ is therefore described as a path through the state-space, with the sum of holding times of states on the path being at least T. Given any path through the state-space, there is a probability that the system's behavior in $[0, T]$ is described precisely by that path.

The system is considered to have failed when certain problem-dependent criteria are met. For example, a critical component may have a spare, but after both the component and its spare have failed the system may be unable to function. A system state reflecting some failure condition is known as a death-state. The tools discussed here estimate the transient probability that the system enters any death-state within a *mission time* T.

We have implemented a parallelized reliability analysis tool based on mathematics discovered by White [19], and two existing tools developed by Butler and Johnson, all at the NASA Langley Research Center. SURE [2] was developed first. It accepts a fully expanded semi-Markov state-space which describes the reliability structure of a given problem, and determines upper and lower bounds on the probability of reaching any model death-state within the mission time T. Each state in the SURE input file is identified by a unique integer (not vector) value. It was quickly recognized that SURE state-spaces are tedious to build by hand, especially given their size and the non-intuitive nature of state identifiers. ASSIST [9] was developed to automatically generate a SURE state-space from a compact and intuitive description of the state space. The reliability modeler describes a system state as a vector of component states, and uses a simple language to describe death-state and search-pruning conditions, to describe conditions under which a particular type of state transition may occur, and how a state vector changes in response following a particular transition. ASSIST accepts the model description, and then generates a file containing the entire SURE state-space. A relatively small ASSIST model can describe a very large SURE state-space.

Both ASSIST and SURE offer many features, perform a great deal of error checking, and present polished interfaces to their users. These tools are in use at approximately forty-five industrial and government sites.

Reliability models with large state-spaces tax both ASSIST and SURE. One problem is simply that of state-space size—models have been known to completely exhaust a 50 MB disk partition allocated to virtual memory. The other problem is computational—the model analysis requires a traversal of every path from the starting state to any death-state. A typical large model may have states numbering in the thousands to tens of thousands and

have an order of magnitude more transitions. Large models create a combinatorial explosion in the amount of work that must be done.

We have developed a tool, ASSURE, that dramatically reduces the time required to analyze ASSIST models. Unlike SURE, ASSURE analyzes system states that are explicitly represented as vectors of state components. Like ASSIST, ASSURE uses the ASSIST language rules to determine whether a given state is a death-state, and to compute the transitions out of the state. The most important innovation of ASSURE over ASSIST→SURE is that ASSURE simultaneously generates and analyzes paths through the state-space. ASSURE's space requirements are dramatically smaller, because ASSURE does not maintain the entire state-space in memory. ASSURE processes a system-state by analyzing it, after which it generates the state's descendents. The memory used used to represent that system state (and the path to it) are then discarded. A second innovation involves our internal representation of a system state, and a path to it. The innovation is made possible by the model analysis mathematics. The heart of SURE (and thus ASSURE) is a theorem that places upper and lower bounds on the probability of the system traversing a given path through the state-space within a given amount of time [19]. When the mission time T is small relative to the mean component lifetime, the formulae comprising these bounds involve sums and products of characteristics of states along the path, implying that one can accumulate these characteristics during the search, instead of saving each individual characteristic. ASSURE is centered around a "path-record" data structure that contains a system state vector and all the accumulated characteristics of some path from the initial state to that system state. The memory required to store a path-record does not change as paths are extended through transitions. ASSURE iteratively removes a path-record from a work-list, determines whether the associated state is a death-state, should be pruned, or generates transitions. Discovery of a death-state prompts calculation of the upper and lower bounds of reaching that state by the path whose accumulated characteristics are recorded in the path record. Generated transitions result in new path-records that are attached to the front of a work list[1]. The memory space for the analyzed path-record is reclaimed, and the process repeats.

Our use of path-records was largely motivated by our anticipated need for dynamic load-balancing. A path-record may be processed on any processor; at any time a processor may share its load by removing a number of path-records from its work-list, and sending them to another processor. The recipient simply inserts them into its own work list. The design decision paid off. Dynamic load-balancing schemes were straightforward to implement, given the simplicity of the work-list and path-record data structures. However, this design decision constrains the utility of ASSURE to models where the mission time is small relative to the mean component lifetime. SURE uses different mathematics to compute bounds for large mission times. This mathematics requires that all characteristics of the given path be individually known, not simply accumulated. Consequently, the longer a path becomes, the more memory is required to save its salient characteristics. ASSURE could be modified to incorporate long mission time analysis; at the time ASSURE was designed

[1]By placing newly generated path-records at the front of the work-list ASSURE is employing a depth-first traversal.

we sought to simply demonstrate that many (not necessarily all) of the problems solved by ASSIST→SURE could be effectively parallelized.

3 The ASSIST Language

The most fundamental idea we use in ASSURE was first exploited by ASSIST: semi-Markov processes of interest can be parametrically described by a simple grammar. To show the power of this idea, we now sketch the main features of the ASSIST language.

Each state in a semi-Markov reliability model describes a possible system state in terms of factors affecting reliability. These states are typically expressible as vectors of integers, e.g., the number of working processors, whether a communication bus has failed (0/1), the number of spare processors. The system makes a transition from a given state when the value of some state vector component changes due to an additional failure or repair.

ASSIST exploits the fact that very many transitions can be described parametrically. For example, imagine that a state has n working processors, each of which has a constant failure rate λ. The rate at which the system makes a transition due to processor failure is $n\lambda$. This single description characterizes a particular type of transition for all values of n. ASSIST recognizes parametric transitions using the TRANTO statement. A TRANTO statement consists of a Boolean conditional which indicates when the transition is permitted to occur, a destination expression which describes how the state is to be modified following the transition, and a rate statement which specifies the transition rate due to the specified transition. For example, a state may consist of a vector <NP,NS,F> where NP denotes the current number of working processors, NS is the current number of spare processors, and F is a flag indicating whether a failed processor is being replaced in this state. Consider the following TRANTO statement:

```
IF F=0 TRANTO NP=NP-1,F=1 BY NP*LAMBDA
```

This type of transition may only occur if no previous processor failure is still being repaired, i.e., when F = 0. In making the transition, the NP component of the state is decremented to reflect the failed processor, and the F component is set to 1 to indicate that a failed processor is being replaced. The cumulative rate at which this transition occurs is NP*LAMBDA, where LAMBDA is the processor failure rate. A preamble in the ASSIST file declares that variables NP and F are components of the system state vector, gives them initial values, and quantifies LAMBDA.

A failed processor may be replaced by a spare. The TRANTO statement

```
IF F=1 AND NS>0 TRANTO NP=NP+1, F=0, NS=NS-1 BY <REPMEAN,REPSD>
```

describes this transition. Here REPMEAN is the mean time of this transition, and REPSD is its standard deviation. This syntax flags the transition as being associated with a recovery process, and gives the information that SURE will need when considering this transition.

Finally, it may happen that a processor will fail while another is being replaced. This is a condition that causes the system to fail, and can be flagged by setting the F variable to 2.

```
IF F=1 TRANTO F=2 BY NP*LAMBDA
```

Ultimately we are interested in the states where the system is considered to have failed. The DEATHIF statement describes such conditions. As we have mentioned, the system is considered to have failed if a co-incident fault occurs. This is indicated with the statement

```
DEATHIF F=2
```

Another type of failure occurs if a processor failed but there are no spares. This condition is indicated with the statement

```
DEATHIF F=1 AND NS=0
```

ASSIST files may also specify that a search be "pruned", meaning that a path is not extended. Pruning is quite important for reducing the complexity of a search, allowing the search to concentrate on paths having the highest probability of traversal. The PRUNEIF statement identifies conditions for pruning a search in the same way that the DEATHIF statement identifies a death-state. One can also prune by probability—prune if the upper bound becomes sufficiently small. A number of other ASSIST features are largely self-explanatory. Figure 1 illustrates a complete, more complex ASSIST model. The system being modeled is composed of three fault-tolerant subsystems; the system is up if and only if all three subsystems are up. Each fault-tolerant subsystem is composed of three components, each having a distinct failure rate. A subsystem is up if any two of its components are up.

The ASSIST syntax is intentionally designed to aid the automated generation of state transitions. Given a state vector we can easily determine (i) whether the state is a death-state (and hence has no transitions), (ii) whether the state meets any pruning criteria (again, no transitions), or (iii) the transitions permitted from the state. For example, consider the processing of the initial state in Figure 1's example. None of the death-state conditions are satisfied, because all `C,S` and `V` components have value 1. The single pruning condition is not met, as `NCF = 0`. However, a number of transitions may occur. Any one of the `C,S` or `V` components may fail, leading to a state vector which is identical to the initial state, except that the status bit for the failed component is now 0, and the `NCF` component is 1. There are nine such states reachable from the initial state. Each of these may reach eight other states, and so on.

In the course of developing ASSURE it became clear that the sparsity of ASSIST's syntax made it difficult to design reliability models of computer networks that incorporate sophisticated recovery algorithms. For example, consider a network that statically routes messages, i.e., the same path is used for every message between a given sender and receiver pair. Suppose that the network has many redundant routing nodes and communication links. The network is temporarily disabled whenever a routing node or link on a static route fails. However, the network can be made operational if a new route using different nodes and/or links can be found to replace the failed one. Therefore, to determine whether the network can be made functional following a link or node failure, we must essentially determine the post-failure network connectivity. Virtually the only way of using ASSIST

```
LAMBDA_S = 7.62E-4;
TIME = 10.0;
LAMBDA_V = 3.9E-4;
LAMBDA_C = 3.5E-4;

SPACE=(C: ARRAY[1..3] OF 0..1,
       S: ARRAY[1..3] OF 0..1,
       V: ARRAY[1..3] OF 0..1,
       NCF);

START = (1,1,1,1,1,1,1,1,1,0);

DEATHIF (C[1]+S[1]+V[1] <2 ) OR
         (C[2]+S[2]+V[2] < 2) OR
          (C[3]+S[3]+V[3] < 2);

PRUNEIF NCF >= 5;

FOR I=1,3;
    IF C[I] > 0 TRANTO C[I] = 0, NCF = NCF+1 BY LAMBDA_C;
    IF S[I] > 0 TRANTO S[I] = 0, NCF = NCF+1 BY LAMBDA_S;
    IF V[I] > 0 TRANTO V[I] = 0, NCF = NCF+1 BY LAMBDA_V;
ENDFOR;
```

Figure 1: An Example ASSIST Model

to test connectivity (e.g. in a DEATHIF statement) is to exhaustively enumerate all of the state conditions under which the network is disconnected. This is clearly unsatisfactory for all but the smallest networks. To deal with this problem we extended the ASSIST syntax.

We will later see that ASSURE translates ASSIST models into C functions. Given this approach it was natural to extend ASSIST by permitting explicit reference to C functions in DEATHIF and TRANTO statements. These functions are permitted to read (and in the case of TRANTO post-conditionals, write) ASSIST system state variables as though they were ordinary C integers or arrays of integers. Instead of constructing Boolean conditionals to identify death or transition conditions, we permit a call to a user-written C function that computes and returns a 0/1 value. In the network example above, we might write a C function `DisConnected()` that determines the network connectivity as a function of the current system state. Extended ASSIST then permits the statement `DEATHIF DisConnected()`. It is permissible to include function arguments in these C functions. Extended ASSIST also permits the user to define and initialize read-only data structures that may be referenced

from within the C functions. For example, this feature is useful for computing and storing the network topology, which otherwise would have to be expressed in ASSIST in terms of constants or one-dimensional arrays of constants.

4 Model Analysis

The mathematics underlying SURE permit one to place upper and lower bounds on the probability of the system traversing a given path within the mission time, T. These bounds depend on a classification of each transition on the path into one of three classes. These classes are explained below, along with definitions needed to express the bounds.

Class 1: Class 1 is composed of failure transitions that occur in states from which there are only failure transitions. Characteristics of this transition are its rate, and the sum of the rates of all other transitions from this state.

Class 2: A Class 2 transition is a recovery transition. Characteristics of this transition are the sum of all failure transitions from this state, the probability that this recovery succeeds over every other recovery transition from this state, and the mean and variance of this transition given that it is taken.

Class 3: A Class 3 transition is a failure transition from a state that also contains a recovery transition. Characteristics of this transition are its rate, the sum of rates of other failure transitions from this state, the probability that the specified recovery transition succeeds over all other recovery transitions from this state, the mean and variance of the recovery transition given that it is taken.

A technically complete description of these transition characteristics can be found elsewhere [2], and is not needed to describe ASSURE processing at a high level[2].

A statement of the SURE theorem in terms that suit our purposes is given below.

Theorem 1 (White) *Let p be a path composed of k Class 1 transitions, m Class 2 transitions, and n Class 3 transitions. The probability $D(p,T)$ of taking a path p through the state-space within the mission time T can be bounded as follows*

$$L(p,T) \leq D(p,T) \leq U(p,T).$$

$L(p,T)$ and $U(p,T)$ have the form

$$L(p,T) = \prod_{j=1}^{k} C_{1j}^{(L)}(p,T) \prod_{j=1}^{m} C_{2j}^{(L)}(p,T) \prod_{j=1}^{n} C_{3j}^{(L)}(p,T)$$

[2] A precise definition of these characteristics and the SURE theorem requires several pages of mathematics whose details are not essential for understanding ASSURE.

$$U(p,T) = \prod_{j=1}^{k} C_{1j}^{(U)}(p,T) \prod_{j=1}^{m} C_{2j}^{(U)}(p,T) \prod_{j=1}^{n} C_{3j}^{(U)}(p,T)$$

where each $C_{ij}^{(L)}(p,T)$ *and* $C_{ij}^{(U)}(p,T)$ *is a function only of the number of Class i transitions,* T*, and the characteristics of the* j*th Class i transition on* p*.*

□

This nice structure of $U(p,T)$ and $L(p,T)$ can be exploited for parallel processing. In the course of expanding p we do not need to store each individual transition's characteristics. We can "accumulate" them as appropriate, thereby saving space. For example, one of the product functions comprising $U(p,T)$ is

$$\frac{\lambda_1 \lambda_2 \ldots \lambda_k T^k}{k!}$$

where k is the number of Class 1 transitions in p, and the λ_i's are their individual rates. When a Class 1 transition with rate λ is taken one need only increment a Class 1 transition counter, and multiply a rate product accumulator by λ. Once a death-state or pruning-condition terminates the expansion of p, then $U(p,T)$ and $L(p,T)$ can be computed from the accumulated characteristic data, and the upper and lower bounds on the system failure probability can be adjusted.

SURE accepts a description of a directed graph representing a state-space. From the initial node S it initiates a depth-first-search of the graph. Whenever a node L without descendents is encountered the SURE theorem is applied to the specific path from S to L. Death-state nodes accumulate the upper and lower bounds of paths that reach them. Once an exhaustive all-paths traversal has been performed one determines the overall upper and lower bound by summing the bounds associated with each death-state node. One can also have the bounds for individually specified death-states printed.

State space graphs may be cyclic, so that an all-paths traversal of the graph will never end. SURE handles the problem by always checking to see if the "next" node in a path is already on the path, forming a cycle. Cycles are "unrolled" for a user specified number of iterations, and the path is carefully truncated using equations developed in [2]. Another form of pruning is to simply terminate a path when the upper bound on the probability of taking the path falls below a user-specified pruning threshold. The overall upper bound on the failure probability is incremented by the path's upper bound, reflecting the worst case scenario where every node immediately reachable from the pruned path represents a death-state. ASSURE uses only the later method. It has been shown that pruning of this type is sufficient to terminate loops[3].

5 ASSURE

ASSURE combines the respective functions of ASSIST and SURE into one integrated tool. At the same time as ASSURE identifies transitions out of a system state (as does ASSIST),

[3]Private communication from Ricky Butler and Alan White.

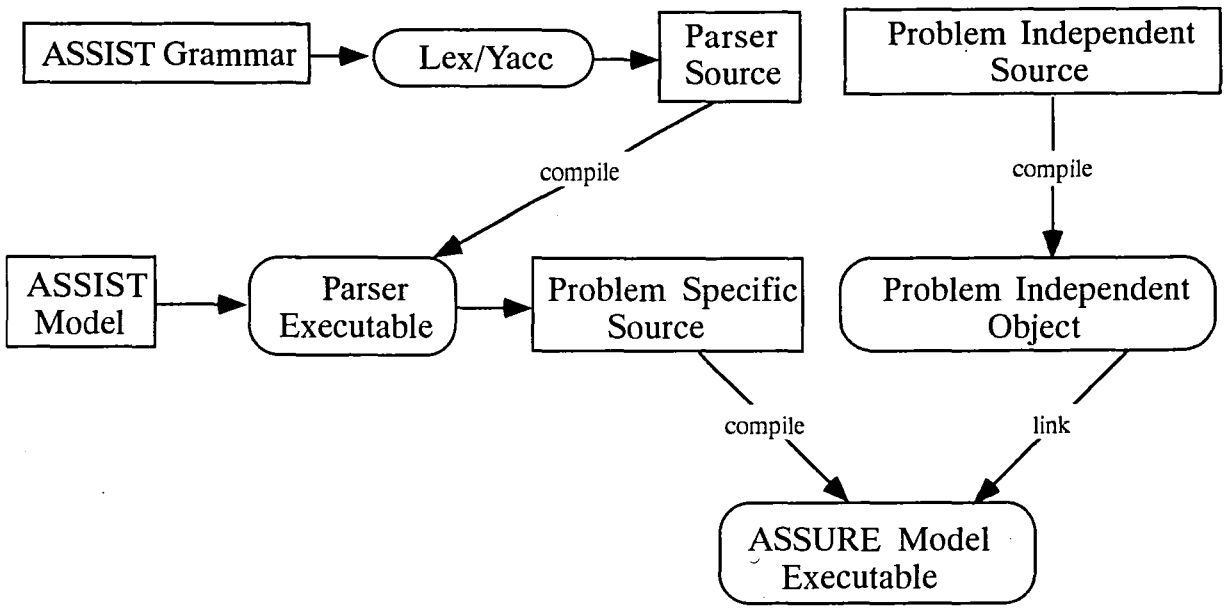

Figure 2: Creating an ASSURE program

it analyzes those same transitions (as does SURE). Not only are we then able to analyze the problem in a single "pass", we achieve substantial savings in memory space as well, permitting ASSURE to analyze problems that defeat ASSIST→SURE. We will later see that excellent parallel run-time efficiencies are also achieved.

We used the UNIX tools Lex and Yacc [10] to describe the ASSIST grammar, and build an ASSIST language parser which transforms ASSIST models into C functions. As a first step, one builds the ASSIST parser, and compiles problem independent control code. This step happens only once. Following it, every ASSIST model undergoes two processing phases. In the pre-processing phase an ASSIST model is parsed, transformed into C functions which are then compiled and linked with the pre-compiled control routines. The pre-processing phase is followed by an execution phase where the problem is solved. These steps are illustrated in Figure 2. The pre-processing and execution phases will next be described in more detail.

5.1 Pre-processing Phase

The pre-processing phase is concerned with the translation of an ASSIST model into C routines that recognize death-states, pruning conditions, and generate transition states. The ASSIST language syntax is closely related to the syntax of imperative programming languages like Pascal and C. It is a conceptually simple matter to transform any Boolean expression of state variables expressed in ASSIST into a corresponding expression in C.

```
int CheckDeath(ptr)
     struct PathRecord *ptr;          /* pointer to a path-record */
{
  if((ptr->State[0]+ptr->State[3]+ptr->State[6]<2) ||
      (ptr->State[1]+ptr->State[4]+ptr->State[7]<2) ||
       (ptr->State[2]+ptr->State[5]+ptr->State[8]<2))
        { AnalyzeDeathState(ptr);
          return(1);
        }
  return(0);
  }
```

Figure 3: Translated C routine to recognize death-states

Therefore, if we can map system state variables onto C language variables we can recognize death-state, pruning, and transition conditions. It is also straightforward to translate ASSIST's looping constructs and the state modification statements following a TRANTO into C language statements.

During the pre-processing stage an ASSIST file is parsed. References to state variables are translated into references to particular offsets within a system state array contained in a path-record. Problem specific C functions are generated for recognizing death-states, recognizing pruning conditions, and generating transitions. ASSURE control code passes a pointer to the path-record of interest to these routines. The routines use the pointer to access individual state variables. Consider the problem expressed in Figure 1. The system state is stored in the ten element array `State` contained in the path-record. Arrays `C`,`S`, and `V` are packed into `State` beginning at locations 0,3, and 6 respectively. `NCF` occupies location 9. Let `ptr` be a pointer to a path record, and `AnalyzeDeathState()` be a routine called when a death-state is recognized. The ASSURE pre-processing stage will produce an integer function `CheckDeath(ptr)` (shown in Figure 3) that can be called to determine if the state pointed to by `ptr` is a death-state[4]. The reader unfamiliar with C can interpret this code by keeping in mind that "struct" variables are basically records, that record fields are accessed through pointers with the `->` symbol, and that `||` denotes a logical OR.

Pruning conditions are checked in entirely the same manner.

If a path-record survives the death and pruning tests it is passed to a routine `TranTo(ptr)` that generates a list of path-records corresponding to the transition states reachable from the state pointed to by `ptr`. `TranTo(ptr)` tests each TRANTO condition; whenever one is satisfied a copy of `ptr`'s path record is made, selected components are modified as described in the ASSIST model; the modified path-record is then attached to the list of transition path-

[4]The code shown in Figures 3 and 4 is equivalent to what is actually produced, but is far more readable.

```
struct PathRecordList *TranTo(ptr)
     struct PathRecord *ptr;
{
 struct PathRecordList *TList;
 struct PathRecord *TranPtr,*CopyPathRecord();
 int I;

  TList = NULL;
  for(I=1; I<=3; I++)
   {
      if(ptr->State[0+I-1] > 0)
       { TranPtr = CopyPathRecord(ptr);    /* make a copy */
         TranPtr->rate = LAMBDA_C;         /* save transition rate */
         TranPtr->State[0+I-1] = 0;
         TranPtr->State[9] = ptr->State[9]-1;
         Attach(TranPtr,TList);            /* attach to tranto list */
        }
      if(ptr->State[3+I-1] > 0)
       { TranPtr = CopyPathRecord(ptr);    /* make a copy */
         TranPtr->rate = LAMBDA_S;         /* save transition rate */
         TranPtr->State[3+I-1] = 0;
         TranPtr->State[9] = ptr->State[9]-1;
         Attach(TranPtr,TList);            /* attach to tranto list */
        }
      if(ptr->State[6+I-1] > 0)
       { TranPtr = CopyPathRecord(ptr);    /* make a copy */
         TranPtr->rate = LAMBDA_V;         /* save transition rate */
         TranPtr->State[6+I-1] = 0;
         TranPtr->State[9] = ptr->State[9]-1;
         Attach(TranPtr,TList);            /* attach to tranto list */
        }
      }
      return(TList);
    }
```

Figure 4: Translated C routine to generate transitions

records. The procedure generated for our example problem is given in Figure 4. Function CopyPathRecord(ptr) allocates a block of dynamic memory for a new path-record, copies the contents of the path-record pointed to by ptr, and returns a pointer to the newly allocated

block. Function Attach(TranPtr,TList) links the newly modified path-record onto a list of transition states generated by the state pointed to by ptr. The expressions used to compute indices into the State vector look curious at first glance. They are the product of a simple translation process that transforms an ASSIST index into a State vector index by adding an offset associated with the state variable, and subtracting one (because C arrays begin with index 0). Any ordinary compiler will combine the constants expressed in the index.

It would be possible to write problem independent routines that *interpret* an ASSIST model, but it seemed to us that the run-time overhead of continually interpreting text would soon exceed the pre-processing cost of parsing the ASSIST file, and compiling the resulting routines. On the realistic problems we study, the pre-processing delay is small.

5.2 Execution Phase

Next we discuss ASSURE's execution phase. First we describe the serial algorithm, and then how it is parallelized.

The execution phase begins once the ASSIST file has been translated, compiled, and linked with problem independent code. Like SURE, ASSURE explores the state space using a depth-first traversal. To start the processing, a path-record describing the starting state is placed in a list of path-records, WorkList. Processing then consists of iteratively selecting a path-record from the front of WorkList, testing the system state S it contains for death-state and pruning conditions, determining all of the states reachable from S and placing path-records for them at the front of WorkList. Processing is complete when WorkList is empty. The serial form of this algorithm is shown in Figure 5. The code is intended to be largely self-explanatory; functions of type void return no values, and the "!" operator applied to an integer is a Boolean 1 if the integer is zero, and is a Boolean 0 otherwise.

ASSURE was designed to permit a straightforward parallelization of the execution phase on a distributed memory multiprocessor. We execute the algorithm above on one processor until the WorkList has at least as many path-records as there are processors. The path-records are then partitioned evenly among all processors, and placed into the individual WorkLists. Each processor then executes the serial algorithm. Whenever a path-record reveals a death or pruned state the path's probability bounds are computed and accumulated in variables that are local to the processor. The fact that a single processor's WorkList goes empty does not imply that the computation is completed (so that the PrintBounds() routine is not immediately called). An idle processor may reseed its WorkList and continue by asking for and receiving any path-record from a processor that has an overabundance of them. The issue of load-balancing is one we will later discuss in more detail. The computation is complete when the WorkList in every processor is empty. At this point one needs to accumulate the upper bounds computed in all processors, and likewise accumulate the lower bounds. These aggregate sums yield the overall upper and lower bounds on the probability of entering a death-state within the mission time.

Key to the approach is the fact that any path-record may analyzed on any processor. One should appreciate the fact that this need not have been the case, and is in fact a

```
AnalyzeProblem()
{
  /* Declaration local variables and forward function references */
  struct PathRecordList *WorkList,*descendents;
  struct PathRecord *path,*Dequeue();
  void Initialize(), ExtendPath(), Enqueue(),
       Release(), PrintBounds();
  int Empty(), CheckDeath(), PrunePath();

  /* Start working */
  Initialize(WorkList);                    /* load starting state */
  while(!Empty(WorkList))                  /* while work to do */
  {   path = Dequeue(WorkList);            /* get first path-record */
      if( !CheckDeath(path) &&
             !PrunePath(path) )            /* if path survives */
       { descendents = TranTo(path);       /* generate descendents */
         ExtendPath(descendents);          /* classify transitions,
                                              accumulate new path
                                              characteristics */
         Enqueue(descendents,WorkList);    /* stick new path-records
                                              at front of WorkList */
       }
       Release(path);                      /* recover memory space */
   }
  PrintBounds();                           /* report the reliability */
 }
```

Figure 5: ASSURE Algorithm

design decision with significant consequences. For example, suppose one needed to know the upper and lower bounds on reaching each individual death-state. This sort of information is easily provided by ASSIST→SURE, but cannot be provided by ASSURE, at least in its present form. Since the state-space is not bound to processors, the contributions to any given death-state's bounds may be distributed among processors. No note is made of the state when the information from a death-state or pruned-state is incorporated into the bounds. Another consequence is that cycles in the state-space cannot be explicitly recognized. Because SURE maintains all the information it may need about a path, it can recognize a cycle and accurately prune paths that endlessly traverse the cycle. ASSURE cannot recognize cycles, because it only accumulates information about a path, it does not

maintain a history of a path's states. Our interest was in seeing if ASSIST→SURE could be parallelized and achieve high performance on an interesting class of problems. We judged that the features we chose not to support were not essential for a demonstration of parallel processing's viability for this application.

Parallel ASSURE exploits parallelism by implementing a parallel depth-first-search for every terminal (i.e., death-state or pruned) node. This type of parallelism has already been well studied [6, 18, 16]; our contribution is to demonstrate that a general tool for an important application class can benefit from parallel processing, without the user having to be involved with the details of the parallelization.

5.3 Dynamic Load Balancing

A driving concern behind ASSURE's design was the recognition that ASSIST→SURE problems are very dynamic in the demands they place on a system, and that dynamic load-balancing would likely be needed to achieve high parallel run-time efficiencies. We next briefly describe the methods we used.

A summary of dynamic load balancing techniques for parallel searching is given in [11]. Methods described there are asynchronous, and local: when a processor becomes idle it polls a small subset of other processors, asking for more work. Our own view on load balancing has been more synchronous and global [13, 12, 14, 15] all processors are involved in every balancing of the workload. Each style has its advantages and disadvantages. The advantage of an asynchronous scheme is that processors with work to do are not delayed by a load-balancing from which they do not benefit. A disadvantage is that it is possible for workload to "pile up" in some localized subset of processors, after which it may take some time for repeated local load-balancing requests to siphon the excess workload from that region. A disadvantage of global schemes is that the per-balance overhead is higher; there is also some doubt about the scalability of global methods. An advantage is that a global scheme treats *potential* balancing problems as well as existing ones. When a global method evenly distributes all existing workload, processors that may soon be empty are given a transfusion of work before it is actually needed. Therefore, one expects that a global load-balancing method will be called less often than a local method. A final advantage follows from the fact that long messages are preferred on distributed memory multiprocessors, as the large fixed communication startup cost is amortized over more bytes. Global methods move more data, and so achieve a lower per-byte communication overhead.

We explored the use of global methods, both because our experience has been in using such methods, and because of the possibility that state-spaces we search may have local concentrations of high workload (this occurs in regions where the modeled system is quickly able to recover from failures.) Global methods are appropriate for quickly breaking these concentrations up.

Two important synchronous load-balancing algorithms have been discussed in the literature. The "dimension-exchange" algorithm [5] assumes that every processor has a number of independent jobs; in d exchange steps it completely balances the workload, d being the

dimension of the hypercube (i.e., the system has 2^d processors). In each step the algorithm balances the workload through one hypercube dimension, as follows. Let $b_{d-1}b_{d-2}\cdots b_0$ be the identity of a processor expressed in binary. In step j this processor balances the workload (i.e., number of jobs) between itself and processor $b_{d-1}\cdots\bar{b_j}\cdots b_0$: the processors compare their loads, and the one with more jobs sheds enough so that each have half of their total. If a processor's load is infinitely divisible, this algorithm is guaranteed to assign exactly the same amount of load to each processor by the end of d steps. Observe that each step actually requires 2 communications per processor—one to inform the partner of its load, the other to send or receive the load.

Load balancing techniques based on parallel-prefix style computations called *scans* have also been suggested [1]. Our adaptation of this method involves two steps. First, every processor P_i submits the length L_i of its `WorkList` to an *enumeration* scan that returns to P_i two values: $S_i = \sum_{j=0}^{i-1} L_j$, and the sum W of all such list lengths. As discussed in [1], one can accomplish this in $2d$ parallel communication steps. Given S_i, processor P_i knows that if all the path-records in the system were enumerated increasingly by processor identity, then its path records would be numbered S_i through $S_i + L_i - 1$. The second step is to distribute the path-records. Since each processor knows the total number of path-records in the system (W), it is simple to evenly remap the path-records as a function of their enumeration indices. For example, processor P_0 will get path-records 0 through $W/2^d - 1$, P_1 will get records $W/2^d$ through $W/2^{d-1} - 1$, and so on. Once a processor knows the indices of its current path-records it can easily compute the identity of processors to whom those path-records should be sent. However, a processor is not able to determine from whom it will receive path-records. If a processor receives an unanticipated message, the operating system keeps the message in system buffer space until a user process asks for it, at which point the system copies the entire message into a buffer indicated by the user process. It is easy to overflow the system buffer space if no message flow control is used. Our implementation deals with this problem by having each processor send short "header" messages to all processors to whom it will send path-records. Since the header messages are short, there is no danger of overflowing the system buffers into which they are initially received. Following the transmission of all such header messages a processor engages in a global synchronization. The processors then logically receive their header messages following the global synchronization, being assured that all such messages are resident in the processor. Thus forewarned, a process can set up receive buffers in the user space for the path-records to follow, and again engage in a global synchronization. Following this synchronization the path-records are sent, received, and placed in the processor's `WorkLists`. As described above the scan-based method requires at least $4d$ parallel communication steps before the path-records are exchanged. It does have the potential advantage that a path-record is transmitted only once—a path-record may be passed as many as d times using the dimension exchange algorithm.

As we will see in the next section, the choice of load-balancing mechanism has only a second-order effect on performance. One method may be significantly faster than the other, and yet the overall performance does not change significantly because load-balancing is needed so infrequently that its cost is not a major contributing factor to the overall

performance. However, we will see that performance is very much affected by whether *any* dynamic load balancing is employed.

6 Performance

We evaluated the performance of ASSURE on a suite of ASSIST models developed at NASA Langley to analyze fault tolerant communication networks. For each model we measure the time required to solve the problem using ASSIST→SURE on a Sun 3/150 workstation, the time required by serial ASSURE on the same workstation, the time required by serial ASSURE on a Cray-2, and the time required by ASSURE on one node, and on 32 nodes of the Intel iPSC/860 multiprocessor. The iPSC/860 is resident at the Institute for Computer Applications in Science and Engineering at the NASA Langley Research Center; the Cray-2 is also at NASA Langley. We find that the parallel version of ASSURE runs three orders of magnitude faster than ASSIST→SURE. Roughly speaking, one order of magnitude can be attributed to the algorithmic improvement of ASSURE over ASSIST→SURE, a second order of magnitude can be attributed to the architectural improvement of an i860 microprocessor over the 68020 used in the Sun 3/150, a final order of magnitude can attributed to the exploitation of parallelism. We find that on the largest problems the parallel version of ASSURE achieves high processor utilizations, and solves the problems faster, by an order of magnitude, than the serial supercomputer. It should be noted that as designed ASSURE is an inherently scalar code, and so cannot benefit from the vector processing capabilities of the Cray[5]; it is not cost-effective to use a Cray as a fast scalar processor. The point we wish to make with the comparison is that traditional supercomputer architectures are not optimal (or even reasonable) for this problem class. On the other hand, distributed memory architectures appear to be ideal for these problems.

The networks in the problem suite achieve reliability through redundancy of network links, and use of dynamic reconfiguration when network components fail. The network fails whenever certain sequences of errors occur in a small enough span of time so that reconfiguration processes are defeated. The different ASSIST models vary in their level of detail; the simpler ones aggregate certain aspects of network behavior, while the most complex one treats it in explicit detail using the extended ASSIST syntax to implement algorithmically expressed state-changes. The networks studied lead to state vectors with thirty to one hundred components; the most detailed network is illustrated in Figure 6. The network connects a fault tolerant processor (FTP) with an array of sensors. The FTP has four channels, accessing the network through one of six network interfaces; each interface is connected to one of two separate network partitions. Only one channel and one network partition is active at a time. The network partitions are comprised of redundant routing nodes and communication links, and serve to connect the channels to clusters of replicated sensors. For each of the reliability models pruning thresholds were selected so that the sum

[5]Any attempt to vectorize SURE would have to go to a great deal of trouble to construct appropriate vectors. It is not immediately obvious if or how this might be done.

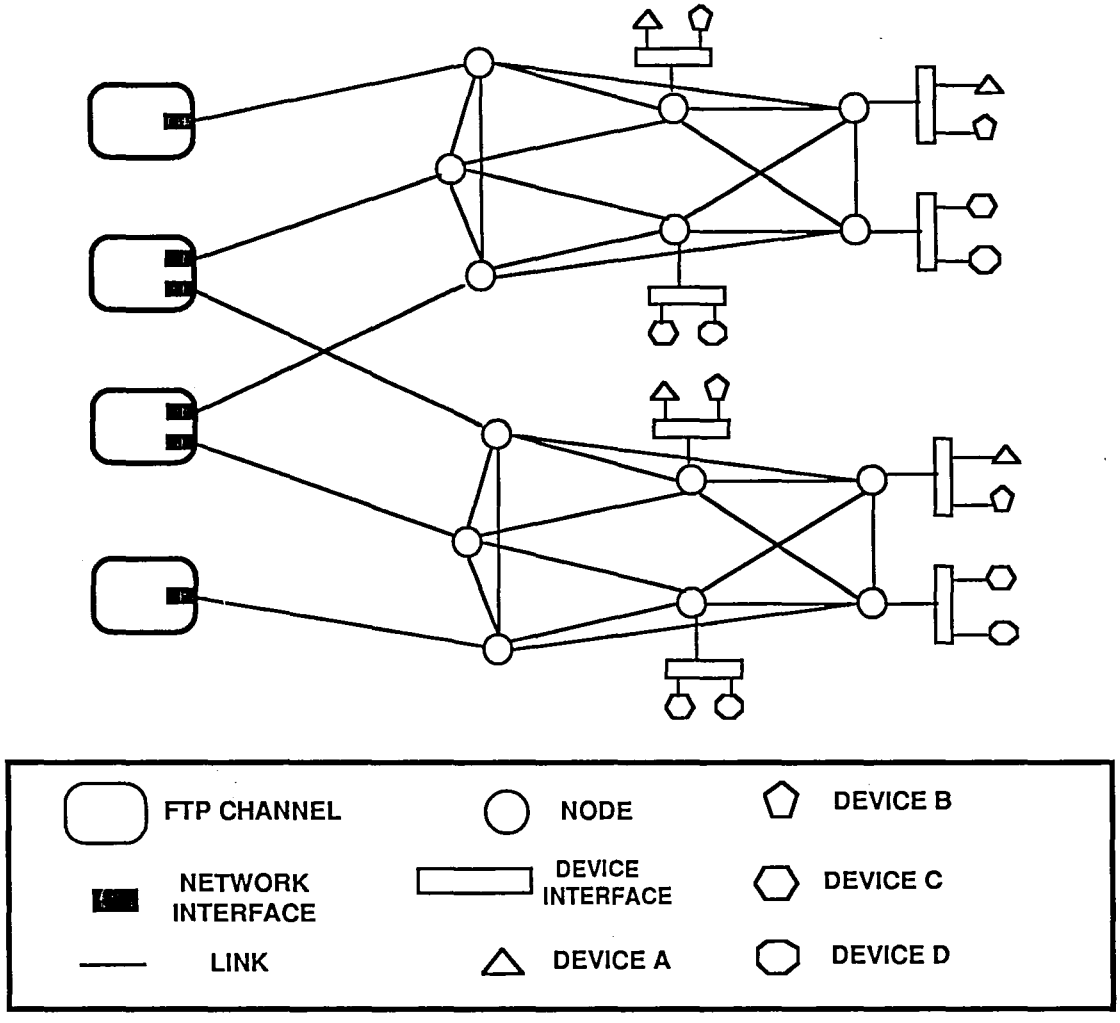

Figure 6: Fault-tolerant network studied

of the upper bounds of all pruned states is at least an order of magnitude smaller than the sum of upper bounds on discovered death-states.

Table 1 gives the number of lines of ASSIST grammar for each model, and the number of path-records analyzed by ASSURE. The number of lines given for Net4 includes the total number of C lines involved in the model.

Table 2 gives the measured performance of ASSIST→SURE and serial ASSURE. We executed ASSIST→SURE on a Sun 3/150 workstation; serial ASSURE ran on the 3/150, one node of an iPSC/860, and a Cray-2. Measurements are on the Sun and Cray-2 are of CPU time; iPSC/860 measurements are of elapsed time. The elapsed time on the Cray

Model	Number of ASSIST lines	Number of path-records
Net1	193	843,378
Net2	269	724,038
Net3	312	3,528,778
Net4	1737 (including C code)	27,204,876

Table 1: Size of model descriptions and number of path-records analyzed

Model	ASSIST→SURE	ASSURE:3/150	ASSURE:i860	ASSURE:Cray-2
Net1	1 hr., 48 min.	12 min.	63 secs.	76 secs.
Net2	1 hr.	10 min.	56 secs.	69 secs.
Net3	11 hr., 30 min.	35 min.	2 min., 40 sec.	2 min., 45 sec.
Net4	N/A	6 hr., 12 min.	39 min., 40 sec.	54 min.

ASSURE's serial performance on network problem suite

Model	NLB (time, utilization)	DE (time,utilization)	SLB (time, utilization)
Net1	(5.5 sec., 36%)	(3.6 sec., 55%)	(4.0 sec., 49%)
Net2	(5.2 sec., 41%)	(3.5 sec., 61%)	(3.9 sec., 55%)
Net3	(9.7 sec., 51%)	(8.2 sec., 61%)	(9.4 sec., 53%)
Net4	(163 sec., 45%)	(88 sec., 85%)	(93 sec., 80%)

ASSURE's parallel performance on network problem suite

Table 2: Serial and parallel performance measurements

timings is at least seven times larger, due to time-shared multiprogramming.

To evaluate the cost and benefits of dynamic remapping we ran each model five times using three different load balancing policies: no load-balancing (except for an initial distribution of path-records to each processor), dimension exchange (DE), and scan-based load balancing (SLB). For both the DE and SLB methods an empty processor broadcasts a request to load-balance. A processor looks for such a message after every 500th path-record it

processes [6]. A load-balance occurs after at least one processor is empty and all processors have recognized the request.

Timings and processor utilizations obtained from these experiments are presented in Table 2. For each set of five runs the performance data shows little variation (with the exception of the number of load balancings on Net4), so that the averages we present are quite typical. For each model we give the time required to run the problem on 32 nodes of an Intel iPSC/860, the average processor efficiency at run-time (this figure multiplied by 32 is the speedup), the average number of load balances required during the solution, and the average time required to perform a load balance once all processors are engaged in the balancing.

One should also consider the time required to translate ASSIST into C and compile the program. The build time using the earlier Intel iPSC/2's compiler is between one and two minutes on all of our network problems. At the time of this writing the linking phase of the iPSC/860 compiler (which runs on the same 80386-based host computer used for the iPSC/2) requires far more time than is reasonable. As we expect this problem to be fixed in the near future we omit exact timings of the build time.

Some features of the parallel performance data in Table 2 and its comparison with the serial performance data are noteworthy. Most importantly, our approach gives the ASSIST modeler the ability to analyze models far more complex than was ever practical using only ASSIST and SURE. High run-time processor efficiencies are achieved on the most complex models. Our results suggest that even higher utilizations will be achieved on larger-scale problems. Indeed, for each of the problems studied here, extremely high utilizations are achieved when we decrease the pruning threshold further, thereby exposing more of the state-space for analysis.

Table 3 reports the average number of load balancings required under the two schemes, and the average cost of performing a balance. The lower per-balance cost of the dimension exchange method can be attributed to fewer communication startups, and a lower volume of communication. There was sufficient variation in the number of load balancings required for Net4 to suspect that the averages given need not be close to the true means.

Despite the apparent superiority of the dimension exchange method over the scan-based method on these problems, the main point to be learned from these experiments is that the method used to balance load matters far less than the decision to support dynamic load balancing at all. Table 2 clearly shows the performance degradation suffered by the no-load-balancing strategy. Load balancing is called so infrequently on the larger problems that its total cost has only a secondary impact on performance.

7 Summary

ASSURE is a tool for analyzing the reliability of embedded computer control systems. ASSURE combines the functions of two prior tools, ASSIST and SURE. In doing so it achieves

[6]The cost of probing for a the existence of a possible message is high on the iPSC/860

Model	Avg. # Balances (DE)	Avg. # Balances (SLB)	Balance Time (DE) (milliseconds)	Balance Time (SLB) (milliseconds)
Net1	34.2	38	10.3	14.8
Net2	39.2	45.6	9.7	14.3
Net3	75.2	71.5	17.1	25.5
Net4	133	109	21.8	35.5

Table 3: Load Balancing Statistics

significant run-time savings over ASSIST and SURE. To reduce run-times even farther, ASSURE has been parallelized on a 32-node Intel iPSC/860 distributed memory multiprocessor. The parallelization is automatic—ASSURE's users need not concern themselves with any details of the parallelization. On a suite of moderately complex models ASSURE achieved very high processor run-time efficiencies. On the iPSC/860 it has solved in eight run-time seconds a model that formerly required half a day on a Sun 3/150 workstation. Furthermore, its input model syntax has been extended to permit the natural construction of models that formerly could not be easily expressed. ASSURE performs especially well on large models, a processor efficiency of 85% is achieved on the most complex (and realistic) model in our suite. Furthermore, ASSURE executes an order of magnitude times faster on these model than ASSURE's serial implementation on a Cray-2. Our development and testing of this tool convincing demonstrates the real viability of using parallel machines to solve complex reliability problems.

Acknowledgements

We thank Ricky Butler, Alan White, and Sally Johnson for their invaluable assistance during this project.

References

[1] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[2] R.W. Butler and A.L. White. Sure reliability analysis. NASA TP-2764, NASA Langely Research Center, March 1989.

[3] G.C. Cohen, C.W. Lee, and M.J. Strickland. Design of an integrated airframe/propulsion control system architecture. NASA Contractor Report 182004, March 1990.

[4] Intel Corportation. *i860 64-bit microprocessor programmer's reference manual.* Intel Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056, 1990.

[5] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[6] R.A. Finkel and U. Manber. DIB: a distributed implementation of backtracking. *ACM Trans. on Programming Lang. and Systems*, 9(2):235–256, April 1987.

[7] R.M. Geist and K.S. Trivedi. Ultrahigh reliability predicition in fault-tolerant computer systems. *IEEE Trans. on Computers*, C-32(12):1118–1127, Dec. 1983.

[8] R.M. Geist and K.S. Trivedi. Reliability estimation of fault-tolerant systems: Tools and techniques. *IEEE Computer*, 223(7):552–62, July 1990.

[9] S. Johnson. The ASSIST language user's manual. NASA Technical Memorandum 87735, August 1986.

[10] B.W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs,NJ, 1984.

[11] V. Kumar and V.N. Rao. Scalable parallel formulations of depth-first search. In *Parallel Algorithms in Machine Intelligence and Vision.* Springer-Verlag, New York, 1990.

[12] D.M. Nicol and P.F Reynolds, Jr. Optimal dynamic remapping of parallel computations. IEEE Trans. on Computers, 39(2):206–219, February 1990.

[13] D.M. Nicol and J.H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Trans. on Computers*, 37(9):1073–1087, September 1988.

[14] D.M. Nicol, J.H. Saltz, and J. Townsend. Delay point schedules for irregular parallel computations. *Int'l Journal of Parallel Programming*, 18(1):69–90, February 1989.

[15] D.M. Nicol and J.C. Townsend. Accurate modeling of parallel scientific computations. In *Proceedings of the 1989 SIGMETRICS Conference*, pages 165–170, Berkeley, CA., May 1989.

[16] V.N. Rao and V. Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[17] H.S. Ross. *Stochastic Processes.* Wiley, New York, 1983.

[18] B.W. Wah and Y.W. Ma. MANIP–a multicomputer architecture for solving combinatorial extremum-search problems. *IEEE Transactions on Computers*, C-33:377–390, May 1984.

[19] A.L. White. Reliability estimation for reconfiguration systems with fast recovery. *Microelectronics and Reliability*, 26(6):1111–1120, 1986.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-182101 ICASE Report No. 90-60		

4. Title and Subtitle	5. Report Date
PARALLELIZED RELIABILITY ESTIMATION OF RECONFIGURABLE COMPUTER NETWORKS	September 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David Nicol Subhendu Das Dan Palumbo	90-60
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18605
	13. Type of Report and Period Covered Contractor Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to IEEE Trans. on Software Engineering

Final Report

16. Abstract

This paper describes a parallelized system, ASSURE, for computing the reliability of embedded avionics flight control systems which are able to reconfigure themselves in the event of failure. ASSURE accepts a grammar that describes a reliability semi-Markov state-space. From this it creates a parallel program that simultaneously generates and analyzes the state-space, placint upper and lower bounds on the probability of system failure. ASSURE is implemented on a 32-node Intel iPSC/860, and has achieved high processor efficiencies on real problems. Through a combination of improved algorithms, exploitation of parallelism, and use of an advanced microprocessor architecture, ASSURE has reduced the execution time on substantial problems by a factor of one thousand over previous workstation implementations. Furthermore, ASSURE's parallel execution rate on the iPSC/860 is an order of magnitude faster than its serial execution rate on a Cray-2 supercomputer. While dynamic load balancing is necessary for ASSURE's good performance, it is needed only infrequently; the particular method of load balancing used does not substantially affect performance.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
reliability, searching, parallel processing, hypercubes	62 - Computer Systems Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	24	A03

End of Document