Final Report : NASA Grant NAG-1-1060

David M. Nicol
College of William and Mary

August 31, 1993

1 Introduction

NASA Grant NAG-1-1060 was initially awarded to study the possibility of using parallel processing to accelerate the simulation of Timed Petri nets (TPNs). It was recognized that complex system development tools often transform system descriptions into TPNs or TPN-like models, which are then simulated to obtain information about system behaviour. Viewed this way, it was important that the parallelization of TPNs be as automatic as possible, to admit the possibility of the parallelization being embedded in the system design tool. Later years of the grant were devoted to examining the problem of joint performance and reliability analysis, to explore whether both types of analysis could be accomplished within a single framework.

In this final report we summarize the results of our studies. We believe that the problem of parallelizing TPNs automatically for MIMD architectures has been almost completely solved for a large and important class of problems. Our initial investigations into joint performance/reliability analysis are two-fold; we have shown that Monte Carlo simulation, with importance sampling, offers promise of joint analysis in the context of a single tool, and we have developed methods for the parallel simulation of general Continuous Time Markov Chains, a model framework within which joint performance/reliability models can be cast. However, very much more work is needed to determine the scope and generality of these approaches.

The remainder of this report outlines the results obtained in our two studies, future directions for this type of work, and a list of publications citing support of NAG-1-1060.

2 Parallelizing Time Petri Net Simulations

There are four fundamental aspects of parallelizing a TPN. First, given an arbitrary net-list description, we must initially partition the net-list into *Logical Processes* (or LPs) that identify parts of the TPN topology that must always be simulated together. Secondly, given a partitioning, we must determine how best to map the LPs onto the multiprocessor architecture. Thirdly, we must be able to dynamically alter this mapping, as there is insufficient

load information to accurately map the simulation prior to runtime. Finally (and not independently), we must be able to synchronize the processors to ensure the correctness of the simulation.

Among the various problems listed above, we have had the least success with partitioning. The essential difficulty is that the fine-grained nature of a net-list description hides underlying structure. For instance, we have simulated many different TPN models of multiprocessor architectures. These models are constructed by replicating a processor model, and connecting replications with a network model. The aggregation models can be very large—one of our models runs to four million places and transitions. The problem of finding embedded *unknown* structure in the net-list, e.g., identifying the replicated processor models, has not been solved to our satisfaction. However, we have found a fall-back position that is acceptable, temporarily. The type of logical aggregation we desire is almost always understood by, and implicit in a user's modeling approach. We need then for a user to *communicate* this aggregation to the parallelization tool. This can be done explicitly, (as we have done in a software tool **pntool** developed under this grant), or may be done more automatically by exploiting an underlying hierarchical structure that is understood by the system development tool.

Assuming that the network has been partitioned into LPs, we must now assign the LPs to processors so as to balance the load and minimize communication costs. Let us for the moment assume that workload and communication estimates are available for each LP. The general problem of mapping workload is computationally difficult, however, for an important subclass of problems it is tractable. Namely, if we enumerate the LPs and restrict ourselves to considering contiguous partitions of that chain, then one can find optimal mappings in low-order polynomial time. In order to avail ourselves of such algorithms, we have looked into the problem of linearizing the LPs. The object of linearization ought to be reduction of probable communication costs under the linear mapping. Since LPs that are adjacent in the ordering can enjoy reduced communication costs by being assigned to the same processor, we wish to find an ordering that keeps highly communicating LPs close to each other. Our solution to this problem is to use "matching" algorithms, as follows. Given a set of LPs, we pair them off in such a way to maximize the sum of edge weights (i.e, communication costs) between paired LPs. Paired LPs are merged into "super-LPs", with edge weights between super-LP's being defined in the natural manner—if LPs A and B are merged and C and D are merged, then the edge weight between the two super-LPs is the sum of edge weights between A and C, A and D, B and C, and B and D. Exact matching algorithms can be expensive, so we have investigated using a linear-time approximation call "stable matching". The resulting mapping algorithm (linearization plus chain mapping) executes quickly enough to be dominated by the execution time of the simulation model. For instance, a very large model (4 million places/transitions) is mapped in about 1 minute, with the bulk of the time cost being devoted to IO processing.

One serious problem is that to map accurately we must have accurate execution and communication cost information. At the time a TPN model is initially loaded we have no measured cost behavior, and are left to estimate these costs from the TPN topology as best

we can. We have discovered that such estimates can be good (if the topology density is in constant proportion to the simulation workload in a region), or can be very poor. Dynamic remapping at runtime is needed to protect against the second possibility. A substantial part of this problem was to restructure the parallel simulation to support dynamic movement of its LPs. The data structures had to be redesigned so that an LP's topological description and marking could be efficiently extracted from a processor and sent to another; in addition, the processor's event list structure had to be localized to provide each LP with its own event list, so that it could also be extracted and moved. Support for runtime reassignment of global place/transition identifiers (i.e., indices in arrays) also had to be provided. Once the support for dynamic assignment was provided, we needed to decide *when* to remap, and *how* to remap. The new mapping construction was treated in two steps. First, runtime estimates of LP event intensities were gathered. To avoid undue overhead we simply measure event *counts*, and use these to compute event rates (per unit simulation time). These rates are the weights used by the mapping algorithm. A new mapping is computed, in parallel, by all processors. We accomplish this by crafting compact codes for each LPs load, and then distribute all codes to all processors, using the vector-OR reduction of the Intel family of multicomputers. We approached the temporal decision problem by applying dynamic remapping decision techniques we've developed before, to the TPN simulation problem. The idea is to periodically recompute a tentative new mapping, and then decide whether to accept and implement it. If the new mapping (based on updated information) appears as though it will reduce the remaining finishing time by at least ten percent, we accept that as positive evidence that remapping is a good idea. However, we don't immediately remap. Instead, we account for the possibility of error, and use the evidence to update a Bayesian estimate of the *probability* that we benefit from remapping. The actual remapping decision policy is to remap when this probability reaches a high level, e.g., ninety percent.

In experimental studies we observe that remapping is indeed required to support good performance, and that the policy is both effective in remapping when it should, and not remapping when it shouldn't.

The problem of synchronizing processors in a parallel TPN simulation can be quite difficult if arbitrary network partitions are permitted. Two fundamental features cause the most trouble. Transitions with zero firing times are one, because a chain of these can set up a causality chain among processors with zero elapsed simulation. In other words, an event that occurs at time t in processor 1 can instantaneously cause an event at t in 2, which can then instantaneously cause an event at t in 3, and so on. We can deal with this problem by simply requiring that all output places of a zero time transition must be assigned to the same LP as is the transition. A second difficulty lies in deciding when a transition is enabled to be fired. To avoid necessitating inter-processor communication to make such decisions, we require that all input places to a transition be assigned to the same LP as the transition, and that all output places of a transition with an input place that is a "decision place" also assigned to the same LP as the transition. A decision place is one that serves as an input place to more than one transition. By placing these restrictions on LP formulation, we ensure that every transition with input places in one LP and output places in other LPs

has the following properties:

- A non-zero firing time, and
- non-preemptable enabling.

These properties ensure that at the simulation instant t when such a transition is enabled, we know with certainty that it will fire, and we know the future simulation instant at which it will fire. This permits us to "pre-send" the effects of the firing to other LPs at the enabling instant, rather than waiting until the firing time arrives. This is lookahead, and is key to a conservative approach to the TPN simulation problem.

For every LP i, among all of its border transitions (those that join LPs) we can identify the one with least firing time, f_i. At any instant we can find the least time stamp t_i, among all pending events on the LP, and know that the LP cannot send a message ever again with a time-stamp larger than $t_i + f_i$. Our synchronization protocol hinges on this observation, as follows. Suppose the processors are all synchronized at time t (they are initially synchronized at time 0). Each LP computes its lookahead bound described above, and the system cooperative computes in parallel the minimum such over all LPs. Call this time $w(t)$. Now the interval of simulation time $[t, w(t))$ has the property that no processor will receive a message with a time-stamp in that window—these have all been pre-sent in previous windows. Hence the LPs are free to execute all their events in $[t, w(t))$, completely in parallel. Upon simulating up to time $w(t)$ (but not including time $w(t)$), the processors ensure that all messages generated in $[t, w(t))$ have been delivered, recompute their lookahead bounds, and compute a new value $w(w(t))$. The next window simulated by the processors is $[w(t), w(w(t)))$, and so on.

Two other approaches permit border transitions to be preemptable. One of these is entirely conservative, and uses the optimistically sent token arrival messages as "appointments" for synchronization. Another approach is optimistic, based on synchronous relaxation. This approach might also allow zero-time transition firing at the border, if necessary. Neither of these variations has been implemented.

A necessary condition for this protocol to do well is that many events be found to simulate in most windows $[t, w(t))$. In our experience this is the case on the types of large TPN models that require parallelization, as opposed, for instance, to models that one can adequate simulate on a set of workstations using independent replications.

3 Joint Performance/Reliability Analysis

Our main thrust for the joint performance/reliability analysis was to determine whether we could perform reliability analysis in a context conducive to performance analysis, using the same basic system model. We answered this in the affirmative, with the tool ASSURE.

ASSURE accepts a model based on the Assure language (which is itself an extension of the ASSIST language) and uses Monte Carlo simulation as the basis for determining relability. The idea is to use the ASSURE model description of transitions and death-conditions to

continuously evolve a system state vector until a death-condition is reached. At this point the SURE theorem is brought into play, being used to find bounds on the probability of the system state taking this particular evolution path within the mission time. Importance sampling plays a critical role in this approach, for without it the occurrence of failures due to co-incident faults will be very much reduced.

The ASSURE language makes it entirely possible to use performance criteria to drive the reliability analysis. Consider a TRANTO statement. In the ASSURE language we may write

```
IF Condition() TRANTO Effect() BY Rate();
```

Here `Condition()`, `Effect()`, and `Rate()` are all C-language functions that have access to the system state vector as C-language variables. The semantic meaning is that if routine `Condition()` returns value true (reflecting the existence of some condition), then routine `Effect()` is called to transform the system state vector as a result. Function `Rate()` gives the transition rate for this particular transition. The key point to appreciate here is the generality of `Condition()` and `Effect()`. One can, for instance, have `Condition()` be a discrete-event simulation that estimates some performance measure that affects reliability, such as CPU workload. For example, if we wanted to model the fact that when the CPU is over 90% utilized then OS software fails with rate λ, then `Condition()` can perform a discrete-event simulation to estimate CPU utilization as a function of the current system state, and so appropriately modify the state vector in `Effect()`. Similar observations apply to DEATHIF conditions determined by C-language functions.

Other features added to the ASSURE package are

- automated parallelization on workstation clusters;
- optimizations for very long mission time scenarios;
- language features for the automated statistical estimation of performance measures in death-states.

User documentation for the ASSURE package is included in this document.

Another area where performance and reliability meet is when a combined performance and reliability model can be expressed as a continuous-time Markov chain (CTMC). For, if we can combine the two features in this framework and simulate the combination with sufficient power, we will have achieved the goal of combined analysis.

We have made substantial inroads into the problem of simulating CTMCs chains on parallel architectures. The key issue we've addressed here is that of synchronization. We have been able to exploit the mathematical structure of continuous time Markov models for synchronization. The basic idea is to recognize that every submodel on every processor is itself a CTMC, and that interactions between processors form a non-homogeneous Poisson process. Knowing this we cause the processors to sample a uniform Poisson process at a higher rate to describe *potential* interactions. Upon reaching a potential interaction point with another processor, a random coin is tossed (with a weight reflecting the disparity

between the current rate of the interaction process and the sampled rate) to determine whether the interaction actually occurs.

The approach outlined above has the very attractive feature that processors may first generate a communication/synchronization schedule, and then perform the simulation adhering to that schedule. Such an ability is unusual in the parallel simulation context, but offers tremendous performance advantages. For instance, we have achieved real speedups of over 220 on 256 processors on the Intel Touchstone Delta architecture.

4 Future Directions for Research

While we are pleased with the results of our research, there remains (as always) further work.

1. There is a great deal of room for good partitioning algorithms for TPNs. The lack of such is probably the largest hole in our TPN work.

2. While ASSURE provides the means of exploring joint performance/reliability models, the level of support provided covered only the tool's development. More work is needed to gain experience with using the tool on joint models, and identifying model features that might inspire further enhancements to the tool.

3. We have shown that the structure of CTMCs can be exploited for accelerated simulation, but are far from being able to provide that capability in a general tool. A good deal more work is needed to find general performance/reliability model descriptions from CTMCs are derived, partitioned, and parallelized. Introduction of importance sampling here may also be critical, owing to the large gap in time-scale between performance-oriented and reliability-oriented events.

5 List of Publications

The following publications cite support of NAG-1-1060.

1. "Rectilinear Partitioning of Irregular Data Parallel Computations", *Journal of Parallel and Distributed Computing*, to appear.

2. "Noncommittal Barrier Synchronization", *Parallel Computing*, to appear.

3. "A Sweep Algorithm for Massively Parallel Simulation of Circuit-Switched Networks", with **Bruno Gaujal** and Albert Greenberg, *Journal of Parallel and Distributed Computing*, to appear.

4. "Optimistic Parallel Simulation of Markov Chains Using Uniformization", with Phil Heidelberger, *Journal of Parallel and Distributed Computing*, to appear.

5. "Parallel Simulation Today", with Richard Fujimoto, *Annals of Operations Research*, to appear.

6. "Conservative Parallel Simulation of Markov Chains Using Uniformization", with **Phil Heidelberger**, *IEEE Trans. on Parallel and Distributed Systems*, to appear.

7. "The Cost of Conservative Synchronization in Parallel Discrete-Event Simulations", *Journal of the ACM*, vol. 40, no. 2, April 1993, 304-333.

8. "Inflated Speedups in Parallel Simulations via `malloc()`", *Int'l Journal on Simulation*, vol 2, Dec. 1992, 413-426.

9. "Performance Bounds on Self-Initiating Parallel Discrete Event Simulations", *ACM Trans. on Simulation and Modeling*, vol. 1, no. 1, pp. 24-50.

10. "Parallel Simulation of Markovian Queueing Networks Using Adaptive Uniformization", with Phil Heidelberger, *1993 SIGMETRICS Conference*, Santa Clara, CA., pp. 135–145.

11. "Parallel Algorithms for Simulating Continuous Time Markov Chains", with Phil Heidelberger, *1993 Workshop on Parallel and Distributed Simulation*, San Diego, CA., pp. 11–18.

12. "Optimistic Global Synchronization for Parallel Discrete-Event Simulations", *1993 Workshop on Parallel and Distributed Simulation*, San Diego, CA., pp. 27–34.

13. "REST: A Parallelized Reliability Estimation System", with Adam Rifkin and Dan Palumbo, *1993 Reliability and Maintainability Symposium*, Atlanta, GA, pp. 436-442.

14. "MIMD Parallel Simulation of Circuit Switched Communication Networks", with Albert Greenberg, Boris Lubachevsky, *Proceedings of the 1992 Winter Simulation Conference*, 629-636.

15. "State of the Art in Parallel Simulation", with Richard Fujimoto, *Proceedings of the 1992 Winter Simulation Conference*, pp. 246-254.

16. "Massively Parallel Algorithms for Trace-Driven Cache Simulations", with Albert Greenberg and Boris Lubachevsky, *Proceedings of the 1992 Workshop on Parallel and Distributed Simulation*, Newport Beach, CA., pp. 3-11.

17. "Parallel Simulation of Timed Petri-Nets", with Subhas Roy, *Proceedings of the 1991 Winter Simulation Conference* , pp. 574-583.

18. "Performance Analysis of Massively Parallel Discrete-Event Simulations", SIGPLAN Symposium on the Practice and Principles of Parallel Programming, Seattle, March 1990, pp 89-98.

19. "Reliability Analysis of Complex Models using SURE Bounds", with Dan Palumbo, submitted to *IEEE Trans. on Reliability.*

20. "Automated Parallelization of Timed Petri Net Simulations", submitted to *IEEE Trans. on Computers*

Automated Parallelization of Timed Petri-Net Simulations*

David M. Nicol†
Weizhen Mao ‡
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795

Abstract

Timed Petri-nets are used to model numerous types of large complex systems, especially computer architectures and communication networks. While formal analysis of such models is sometimes possible, discrete-event simulation remains the most general technique available for assessing the model's behavior. However, simulation's computational requirements can be massive, especially on the large complex models that defeat analytic methods. One way of meeting these requirements is by executing the simulation on a parallel machine. This paper describes simple techniques for the automated parallelization of timed Petri-net simulations. We address both the issue of processor synchronization, as well as the automated mapping, static and dynamic, of the Petri-net to the parallel architecture. As part of this effort we describe a new mapping algorithm, one that that also applies to more general parallel computations. We establish certain analytic properties of the solution produced by the algorithm, including optimality on some regular topologies. The viability of our integrated approach is demonstrated empirically on a large scale parallel architecture, where excellent performance is observed on various models of parallel architectures.

---

*A preliminary version of this paper appears in the Proceedings of the 1991 Winter Simulation Conference under the title "Parallel Simulation of Timed Petri Nets".

†This work was supported in part by NASA Grant NAG-1-1060, the Army Avionics Research and Development Activity through NASA grant NAG-1-787, NASA Grant NAG-1-1132, and NSF Grants ASC-8819373, and CCR-9201195.

‡This work was supported in part by NSF Grant CCR-9210372.

1 Introduction

Timed Petri-nets (TPNs) are an important modeling tool used to study the behavior of various types of complex systems. While a great deal of study has gone into the analytic properties of TPNs (e.g., see [16] and its references), in practical settings TPNs are generally simulated. For example, simulation of TPN-related models is the basis for the performance analysis in ADAS[7], a tool designed specifically for parallel hardware and software performance evaluation. Discrete-event simulation of TPNs is thus an important modeling and analysis activity, and is one known to require a great deal of computational effort. Parallel execution offers the possibility of decreasing the execution time of TPN simulations; however, the user community will adopt parallelized simulations only if the parallelization is largely automatic. The problem of automating such parallelization is the topic of this paper. In particular, we describe methods for synchronizing processors in a parallel TPN simulation, and for load-balancing parallel TPN simulations. Our methods have been implemented in a tool where one designs the TPN graphically, after which all parallelization is handled automatically. The target architecture for this tool is the Intel family of multicomputers. Good performance (e.g., speedups greater than 40 using 64 processors) has been observed on large TPN models of parallel architectures, including nearest neighbor meshes, slotted rings, and Thinking Machines CM-1 global routing network.

Parallelized discrete-event simulation has been actively studied over the last ten years; the survey in [8] is an excellent introduction to the topic; a newer survey [21] highlights current areas of research interest. Synchronization between processors has been and remains a subject of much interest, owing to the complexity of the synchronization requirements imposed by discrete-event simulations. The difficulty arises from the fact that the simulation model is typically partitioned among processors, each of which maintains its own simulation clock. An event associated with the submodel assigned to one processor may affect some portion of a submodel assigned to another processor, thereby necessitating an interprocessor communication. A parallel discrete-event simulation can be viewed then as a collection of communicating discrete-event simulations of submodels. In this context the notion of simulation time imposes synchronization requirements. Consider: an event (e.g. a message passed by a simulated PE (processing element)) occurs at simulation time s on some processor, and affects the submodel on another processor (e.g. the message arrives at another PE), say at time $s + d$. If the affected processor has already past time $s + d$ it may have done so incorrectly as it has neglected to consider the effect of the message arrival at time $s + d$. Synchronization protocols deal with this problem. Two fundamentally different styles of protocols have been studied. *Conservative* approaches (e.g. [4, 15, 23]) ensure that a processor does not advance its simulation clock until it is certain that it will not bypass some simulation time at which another processor affects it. Conservative protocols are known to require *lookahead* in order to avoid deadlock, and to achieve good performance. Lookahead is the ability of a processor to predict its future behavior, as regards when next (in simulation time) it may affect another processor's submodel. *Optimistic* approaches ([11]) permit a processor to simulate ahead under the anticipation that another processor will not affect its submodel in the "past", but then correct these temporal

errors as they occur. Optimistic approaches require state-saving and rollback to function properly. The notions of conservatism and optimism are not mutually exclusive; as observed in [29], the space of synchronization protocols is better partitioned using finer distinctions. This leads to protocols that combine elements of optimism and conservatism.

The synchronization approach we develop in this paper is conservative in all respects. A principal contribution of this paper is to demonstrate that effective automated parallelization of TPN simulations is possible using a very conservative, very simple, synchronization scheme. The lookahead calculation is easy and automatic, and we provide a new automated mapping algorithm with a demonstrated ability to balance workload and keep communication overhead low. We also incorporate dynamic remapping logic, and observe how it substantially boosts performance.

The parallelized simulation of TPNs has not received much attention. This is due in part to the fact that the conceptual model of parallel simulations that is usually studied (based on the seminal work in [4]) precludes Petri-net semantics, an observation detailed in [36]. This conceptual model ascribes fixed communication channels between *logical processes*; time-stamped messages are exchanged via these channels, and an LP's simulation clock is advanced as a result of consuming a message. The solution described in [36] involves extension of this model to support Petri-net semantics. SIMD simulation using recurrence relations of a constrained class of stochastic TPNs is developed in [2]. In work more closely related to ours, Sellami and Yalamanchili [32] and [33] consider a conservative protocol to simulate "marked graphs", which are derived from a restricted class of TPNs. They too exploit model characteristics to optimize the synchronization protocol and to partition the marked graph model. The conceptual model we have most recently used [23, 25, 24] is simply that of communicating discrete-event simulations. Our model employs the same semantics of event list manipulation as does traditional serial discrete-event simulation, and so does not suffer from the limitations of the message-consuming model. However the specifics of our synchronization protocol require that some care be taken when partitioning a TPN among processors. In extreme cases these requirements may preclude any parallelization by our methods. We believe these cases are unusual, especially in TPN models of parallel architectures. One simple condition that ensures our protocol will work is if every communication between distinct "modules" (e.g. PEs, memories) in a simulated architecture is modeled with a transition having a non-zero firing time. This simply models the real world constraint that communication takes time.

A simulation's workload cannot in general be predicted in advance of actually performing the simulation. Consequently a parallelized simulation must be prepared to measure workload at runtime, and dynamic remap if needed to balance it. Early work on the problem was developed in [22]. The basic idea is to measure multiple trial runs, analyze critical path information from each, and cluster pieces of the simulation model based on aggregated critical path information. Later work done developing the Time Warp Operating System (TWOS) [12] approached the problem with multiprocessor scheduling heuristics, with workload measured as "effective utilization". The optimistic nature of TWOS synchronization adds interesting dimensions to the load-balancing problem. Similar ideas are explored in [9], [34], save that the notion of workload is slightly different. These methods rely upon a centralized process to compute and distribute new load distributions. Their

rebalancing algorithms typically consider incremental movement of LPs in efforts to reduce the total communication cost.

Another recent line of inquiry is based on heuristic graph partitioning, e.g., [17], [30] and their references. Here one seeks to aggregate nodes (elemental pieces of the model) into equal-sized blocks, so as to minimize the sum of edge (e.g., communication) costs between nodes assigned to different blocks. This problem formulation has a large literature in the VSLI design community. While the approach has been applied to the mapping problem in parallel processing, and to discrete-event simulation in particular [17], we have chosen a different approach for two reasons. First, one can rarely analytically quantify the quality of a graph-partitioning solution, except to assert that the solution cannot be improved over any set of small local exchanges between blocks. The approach we develop has some analytic assurances. Secondly, the objective of minimizing the total sum of all communication costs does not capture the fact that communication is parallelized. In our approach we seek to minimize an objective function that better models execution time.

The contributions of this paper are two-fold. First, we develop a new heuristic for static mapping, and analytically quantify certain aspects of the solutions produced. In some cases we can bound the deviation of the results from optimal, in other cases we can prove optimality itself. Secondly, we extend pre-existing work in synchronization and also in dynamic remapping decision-making to the TPN simulation problem, and synthesize these adaptations with the new mapping algorithm. The resulting system is capable of accepting a TPN model designed graphically and without explicit concern for parallelization, then automatically mapping, synchronizing, and dynamically remapping the simulation executing on a large scale parallel architecture. We prove the feasibility of this automated approach by demonstration, simulating a number of large TPN models, including one of the Thinking Machines CM-1 global routing network, and a slotted-ring parallel architecture. Good performance, obtained automatically from graphical TPN descriptions is reported for these simulations on large scale parallel architectures. In one case we observe a speedup in excess of 43 on 64 processors of the Intel Touchstone Delta.

The remainder of this paper is organized as follows. In Section §2 we discuss how TPNs work. Section §3 develops our synchronization and simulation algorithms. Section §4 discusses the automated mapping algorithms we use. Finally, Section §5 presents the performance results of simulating several parallel architectures. Section §6 summarizes this paper.

2 Background

A Petri-net can be viewed as a bipartite graph, with each node classified as either a *place*, or a *transition*. The usual graphical conventions depict a place by a circle, and a transition by a straight line. Places may direct arcs to transitions, and transitions may direct arcs to places. Each place that directs an arc to a transition t is known as one of t's *input places*; likewise, each place to which t directs an arc is known as one of t's *output places*. Input and output transitions are similarly defined with respect to a place. A place may hold any number of *tokens*; the tokens may move from place to place in accordance with the *transition firing* rule. A transition t may *fire* if each of

its input places has at least one token each. The effect of t's firing is to remove one token from each of t's input places, and to add one token to each of its output places.

A place with more than one output transition is known as a *decision place.* The arrival of a token at a decision place may fulfill the firing requirements of more than one transition. However, only one of these transitions should actually be permitted to fire, as the firing of the first such will remove the enabling token from the decision place. A standard means of resolving this dilemma is to non-deterministically choose which transition (among those able to fire) will actually fire.

An ordinary Petri-net has no notion of "time". A common variant of timed Petri-nets associates time with transition firings, as follows. Suppose the conditions to fire a transition are met at time s, and the *firing time* associated with that transition is δ. Then

- At time s, one token is removed from each of t's input places;
- From time s to time $s + \delta$ the transition is considered to be *firing*;
- At time $s + \delta$ a token is added to each of t's output places.

We say that the transition firing is *enabled* at time s, and *completes* at time $s + \delta$. Note that tokens are committed to the transition firing at the time of the transition being enabled, not at the point when the transition actually fires. The interpretation that commits tokens upon firing is also common; we will later discuss how our synchronization protocol is able to handle this variation as well.

Using the rules above, we may construct a discrete-event simulation of a TPN whose events are **TokenArrival**, **BeginFiring**, and **EndFiring**, which denote the arrival of a token to a place, the beginning of a transition's firing, the removal of a token from a place, and the ending of a transition's firing, respectively. Assuming that the initial marking of tokens to places appropriately initializes the event list with **TokenArrival** events, the simulation may be implemented using the following sequence.

1. Fetch the next event from the event list, say with time T_{sim}. Advance the simulation clock to time T_{sim}.
2. Execute the event, in one of the following manners.

   Case: TokenArrival Let p denote the associated place. Increment the token count at p. If the previous token count was non-zero, then the event processing is finished. Otherwise, this token's arrival may enable the firing of some transition. In this case, among all of p's output transitions, identify those now enabled to fire due to the token's arrival. Choose one of these uniformly at random, say t, and insert a **BeginFiring** event for t in the event list, with time-stamp T_{sim}.

   Case: BeginFiring Let t denote the associated transition, and let δ_t denote its firing time. Decrement the token count at each of t's input places. For every one of t's output places p', insert a **TokenArrival** event with time-stamp $T_{sim} + \delta_t$ into the event list. Finally, insert an **EndFiring** event with time-stamp $T_{sim} + \delta_t$ into the event list.

Case: EndFiring This event is used only to permit the measurement of any statistics desired at this time by the modeler. No additional events are scheduled by processing an **EndFiring** event.

3. Return to step 1 if termination conditions are not met.

It may seem curious that we insert **TokenArrival** events into the event list as a result of **BeginFiring** processing, instead of **EndFiring** processing. We deliberately formulated the solution this way in order to highlight the lookahead that exists in TPN simulations—at the time a transition begins its firing we can predict exactly when tokens generated by the firing appear in their new places. Our parallel solution will exploit this fact. Lookahead of this type is not necessary in purely serial simulations.

As discussed in [16], there are a number of ways one can augment this basic structure of TPNs, some of which we have used in our architectural models. Some arcs may *inhibit* rather than enable transitions, which means that the associated place must be empty for the transition to fire. Another modification allows one to specify a priority ordering on output arcs from a decision place, to give some control over which transition might be enabled. Yet another allows one to associate a probability distribution with the output arcs of a transition—on firing, one randomly chosen output place receives a token. Additional modifications include the association of colors with tokens, and allowance for an arc to carry more than one token when it fires. All of these have important applications, and can be incorporated directly into the framework we propose.

In the section to follow we show how to implement this algorithm on a parallel computer.

3 Synchronization

We anticipate that parallel simulation will be practical primarily when large simulation models are distributed over a moderate number of processors. The usual use of discrete-event simulations is to construct confidence intervals from simulation output. Confidence intervals call for independent replications, and there is scarcely any easier way to exploit parallelism than to concurrently run independent replications. However, one rarely wants to run more than, say, twenty replications of a long-running simulation, because the width of a confidence interval decreases only in proportion to the inverse square root of the number of replications. The implication is that given a 500 node multiprocessor, one is more likely to devote 25 processors to each of 20 independent replications than one is to devote an independent replication to each processor. It is also frequently the case that simulation is used as an exploratory tool, where a decision is made on the basis of one run. It may be useful then to execute that run as quickly as possible, on one of the widely available parallel systems. Thus we believe that techniques for parallelism have practical interest. We also believe that parallel simulation will be useful primarily on large simulation models. Small simulation models are simulated sufficiently quickly on workstations or PCs. Parallel architectures offer increased main memory size over conventional architectures, which helps to avoid running the simulation in virtual memory and its attendant paging costs.

For the reasons outlined above we have concentrated on parallel simulation techniques suitable for large simulation models. We have studied a conservative synchronous approach to synchronization and demonstrated analytically that it can achieve good performance when the size of the simulation model is large [25, 24]. The solution we now develop for TPN simulations is an application of this approach to the TPN problem. We extend our previous work by tailoring the approach to work around TPN features that cause difficulty for parallelized simulation, and to take advantage of TPN features that ease parallelized simulation.

The remainder of this section is divided into three parts. The first part provides some general information needed to understand the synchronization issue. The second part discusses the basic synchronization itself, while the third part extends the method to TPNs where transition firings may be preempted.

3.1 Preliminaries

We assume that the TPN model is partitioned by the modeler into logically cohesive subnets we call *Logical Processes*, or simply, LPs. LPs are mapped to processors. Every processor maintains its own simulation clock, and an event list for every LP. A min-heap maintained over the minimal elements of each LP's event list allows us to treat the processor as having a single event list. One processor communicates with another by sending a time-stamped message. In our framework that message always reports the arrival of a token to some place, and the time-stamp records the arrival time.

All places and transitions in a given LP will always be executed on the same processor, even if the LP's processor assignment changes. The problem of effectively aggregating a netlist description of a TPN into LPs seems to be extraordinarily difficult, and perhaps unnecessary. Petri nets are commonly developed using graphical tools; these tools frequently let the modeler aggregate and then duplicate some subnet, e.g., a processor or an interface logic module. These are excellent candidates for LP aggregation, and it is a simple matter for a modeler to graphically communicate such aggregation to the tool. We have done exactly this in our tool **pntool**[20], that serves as the graphical front-end for our automated TPN parallel simulation testbed.

Our solution requires that two rules be followed when aggregating (and **pntool** enforces these rules).

- All input places for a transition are assigned to the same LP as the transition.
- Every transition t with an output place that is assigned to a different LP than t must have a non-zero firing time.

The first rule vastly simplifies the logic needed to decide when a transition may fire. Alternate synchronization schemes for timed Petri nets do not make this assumption [36, 13]. The second rule ensures that there is always a lapse of simulated time between when a transition firing on one processor may affect the state of another processor. This requirement could be relaxed (with some modifications to our approach), but at the cost of increased synchronization communication. We

have chosen not to do so, both for the purposes of increased performance, and because we feel it is natural to have non-zero time transitions between LPs.

The event processing logic on every processor is identical to the event processing described in Section §2, save that the code must detect when to send a **TokenArrival** message rather than insert a **TokenArrival** event in an event list. Our synchronization protocol establishes control over these inter-LP message communications. The protocol relies on two key activities: the *pre-sending* of **TokenArrival** messages, and the computation of lower bounds on the time-stamp of the next message an LP might send to an off-processor LP. We have already introduced the notion of pre-sending **TokenArrival** messages—these messages are sent as part of **BeginFiring** event processing, rather than **EndFiring** event processing.

Given that **TokenArrival** messages are pre-sent, one can, at any time, compute a lower bound on the time-stamp of the next message a processor may send to another. For any LP, consider the set of *border* transitions, those transitions assigned to it which have output places assigned to a different LP, on a different processor. Let δ_{min} be the minimum firing time among all border transitions in all LPs. Now suppose that T_{sim} is the value of a processor's simulation clock after completing some event's processing (the lookahead we discuss here is computed between the processing of events, not during). The next message sent off-processor cannot have a time-stamp smaller than $T_{sim}+\delta_{min}$, for only **BeginFiring** events send messages, and the time-stamps on these messages are constructed by adding the processor's clock value to the transition's firing time. Thus, $T_{sim} + \delta_{min}$ always provides the desired upper bound. A potentially larger *conditional* bound can be constructed with very little extra cost by replacing T_{sim} with the least time-stamp on any event in the event list, say E_{min}. We take $E_{min} = \infty$ if the list is empty. The validity of this bound is conditioned on the processor not receiving a **TokenArrival** message with a time-stamp smaller than E_{min}. We have shown in [25] that bounds conditioned on the absence of further message arrivals suffice for our protocol. Our parallel solution assumes the existence of a routine **BoundNextMsgTime()** that finds E_{min} and returns the sum $E_{min} + \delta_{min}$. We turn next to a discussion of the protocol and its integration into the simulation algorithm.

3.2 Parallel Algorithm

The following is a brief overview of the protocol. Suppose that all simulation events in all processors up to (but not including) time T_{sim} have been simulated. Our protocol will compute a simulation time $w(T_{sim})$, such that all events with time-stamps in the *window* $[T_{sim}, w(T_{sim}))$ can be executed without further communication between processors. The openness of the upper window edge is deliberate, in order to avoid the receipt of the message with time-stamp $w(T_{sim})$ which defines the window. Off-processor messages generated in the course of processing **BeginFiring** events may be buffered and delivered at the end of the window processing; alternatively, they may be sent and received directly. Upon receipt a message is converted into a **TokenArrival** event and is inserted into the recipient's event-list. Messages sent between co-resident LPs are converted immediately into events. If programmed properly, there is virtually no additional overhead due to on-processor inter-LP communication. Once all events up to time $w(T_{sim})$ are known to have been simulated, a

new window is computed, and the process repeats.

From the description above it is clear that the time $w(T_{sim})$ must be chosen carefully. Given that all processors have simulated all events up to time T_{sim}, $w(T_{sim})$ is computed by having each processor call **BoundNextMsgTime()**. $w(T_{sim})$ is defined to be the minimum conditional bound returned, among all processors. The global minimum computation can performed in $O(\log P)$ time on most multiprocessors, where P is the number of processors. We have proven elsewhere [25] that every off-processor message the simulation will send after this point has a time-stamp of at least $w(T_{sim})$. Thus, all inter-processor messages that the simulation will generate in the interval $[T_{sim}, w(T_{sim}))$ have already been identified, and converted into events. Every processor may therefore simulate its submodel up to (but not including) time $w(T_{sim})$ without danger of receiving a "late" message. This property leads us to the protocol given below.

1. For every initial token, insert a **TokenArrival** event in the appropriate processor's event queue, with time-stamp 0.

2. Set $s_1 = 0$, set $i = 1$.

3. For every processor, call **BoundNextMsgTime()**. Use a logarithmic time min-reduction to compute $w(s_i)$—the minimum value returned by any **BoundNextMsgTime()** call. Every processor learns the value of $w(s_i)$.

4. Every processor may now simulate its submodel up to time $w(s_i)$, independently of and in parallel with all other processors. Event processing is identical to that described in Section §2, save that **TokenArrival** events destined for off-LP places are passed as time-stamped messages. Messages between co-resident LPs are converted immediately into events.

5. The processors synchronize globally. Following the synchronization, every processor accepts any remaining unreceived messages sent to it during the processing of window $[s_i, w(s_i))$. A processor consumes a received message simply by inserting the described event into its event list.

6. Define $s_{i+1} = w(s_i)$, then increment i. Check termination conditions, return to step 3 if the termination conditions are not satisfied.

The reason for this protocol's success on large models is quite intuitive. Imagine the simulation time line, and mark it wherever an event occurs. This protocol slides a window across the time-line, allowing processors to execute simulation workload in parallel during the span of a window. Each window is at least $\delta_{\min}$ time units long. As we increase the size of the simulation model (presuming $\delta_{\min}$ does not also increase), the density of events on the simulation time line will increase, and so the number of events within a window will increase. The overhead of the protocol lies only in determining the size of the window; hence, as the model size grows the protocol's overhead is amortized over an increasing number of events. Of course, one still strives to keep the overhead low, but it is reassuring to know that regardless of its cost, it can be spread over the processing

of many, many events on large models. The protocol can identify many events even when there is no minimum firing time. Applying the results of [25] to TPN, we are assured that if firing times are random and exponentially distributed, then the number of events in a window grows without bound as the model size is increased.

3.3 Tokens Committed at Firing

We are also able to handle TPNs with a different firing rule: if a transition with firing time δ is first enabled to fire at time s, and remains enabled throughout time interval $[s, s+\delta]$, then (and only then) the firing occurs at time $s+\delta$ and token counts are adjusted at the transition's input and output places. Due to the influence of decision places, we cannot commit to the effects of firing a transition until the full enablement duration has elapsed. For instance, suppose transitions t_1 and t_2 share a input decision place p. A token arrives at p and enables both of them. The first to fire consumes the token at p, and so disables the other. A serial simulator would post **EndFiring** events for both transitions at the instant they become enabled. Then, the first **EndFiring** event processing will include a re-analysis of the status of all transitions that might have been disabled; the **EndFiring** event for each such is removed from the list. Consequently, in the parallel simulation one cannot safely pre-send **TokenArrival** messages for preemptable transitions.

The easiest solution is to prohibit errant messages from being sent. We simply constrain LP formulation further so that if t is a transition with a decision input place, than all of t's output places are assigned to the same LP as t. This rule ensures that erroneous **TokenArrival** arrival messages are never sent, since nothing can interfere with the firing of a border transition once it is enabled.

The solution above may cause a model to be so over-aggregated so that opportunities for parallelism are limited. If this is the case, it is possible to deal with the situation in a number of ways; however all require some increased communication. One method is to have **TokenArrival** messages be sent as before, recognize that those from transitions with decision input places are *tentative*. Tentative **TokenArrival** messages are treated as "appointments" [23]. A processor Q receiving one with time-stamp s from processor P will not simulate any event with time-stamp s or greater until it is given permission by P. If P ends up canceling that **TokenArrival** event, it immediately sends a message to Q notifying it to cancel the event. If P ends up actually simulating the associated **EndFiring** event, then it sends a message to Q indicating the **TokenArrival** event is correct. Deadlock is avoided if one ensures that there is a positive delay $\delta > 0$ between when the receipt of a **TokenArrival** message to an LP can affect the behavior of any of its border transitions.

If event cancellations are rare, then we may be able to avoid the additional message-passing and synchronization costs of appointments by using optimism. It is not difficult to synthesize our approach with the method of *synchronous relaxation* [6]. The idea here is to simulate a window as before, with pre-sent **TokenArrival** messages, which we assume are correct. We can always detect when a tentative message was incorrect, because the sending processor will simulate the disenablement of the transition whose firing was already reported. That "error" is easily corrected by sending an event cancellation message after the erroneous **TokenArrival** message. The window

is resimulated then. If more errors are discovered in the next pass, then they are repaired and the window is resimulated again. This process continues until the window is simulated without error. Convergence is assured because the correction to the earliest fault in an iteration cannot be undone by any later iteration. State-saving is required in order to support a window's resimulation. This, and the cost of repeating a window's simulation are the main overheads of the method.

One of the attractions of using synchronous relaxation with our windows is that we can take advantage of window properties to minimize the communication overhead. In the general synchronous relaxation method a correction may end up causing the transmission of a message that has never been seen before. This has nontrivial ramifications on the types of error corrections that have to be anticipated. However, we can use synchronous relaxation so that the only issue to be resolved by iteration is the validity of tentative inter-processor **TokenArrival** messages.

We would like to minimize the communication cost of message cancellation, and so modify the protocol so that any message with time-stamp s' is buffered until the simulation reaches the window $[s, w(w))$ containing s'. The message is sent just prior to simulating $[s, w(s))$. Holding back the message this way does not affect the value of $w(s)$ computed (because the **TokenArrival** message being withheld has a corresponding **EndFiring** event with the same time-stamp in the sending processor's event list). This arrangement permits a processor to cancel any tentative message locally (i.e., without interprocessor communication) in any window prior to the one containing its arrival time. Of course, if sent, the message might still be cancelled by some event between times s and s'. The processor P that originally sent the message to Q discovers the cancellation conditions, and sends a cancellation message after it. Receiving the cancellation, Q changes the validity status of the message, recovers its state at s, and resimulates the window.

The algorithm executed by each processor is presented below. We presume that every tentative message has a "valid/invalid" bit. It is possible for an event cancellation message to be itself canceled, so we define the effects of a cancellation on a tentative message to be an inversion of its valid bit. In the description below, set *Active* contains all tentative messages sent by the processor in the present window, and *Buffered* holds all known messages generated by the processor which have not yet been sent. Cancellation message generation is handled simply. During any iteration, the processing of an **EndFiring** event for transition t checks to see if (i) an associated tentative **TokenArrival** message in *Active* or *Buffered* was considered to be valid or not in the previous iteration, and (ii) whether any tentative **TokenArrival** message in *Active* or *Buffered* cancelled by this event was considered to be valid in the previous iteration. These checks are performed on status bits of messages in *Active* and *Buffered*. The effect of a conflict between the valid bit and the simulation state is to invert the valid bit. If the errant message is in *Active*, a cancellation message is sent to the message's recipient.

The algorithm's description follows.

1. For every initial token, insert a **TokenArrival** event in the appropriate processor's event queue, with time-stamp 0. Assign $s_1 = 0$, $i = 1$, *Buffered*=$\emptyset$, and *Active*=$\emptyset$.

2. Compute and distribute $w(s_i)$.

3. Move all messages in set *Buffered* with time-stamps less than $w(s_i)$ into set *Active*. Set the valid bit in each message in *Active*, and send copies of all messages in *Active* to recipient processors.

4. Synchronize, and store received messages. Call this set *Received*. Checkpoint state.

5. Convert all valid messages in *Received* into events.

6. Every processor independently simulates its submodel up to time $w(s_i)$. Generated messages to be sent off-processor are placed in *Buffered*. Validity of messages in *Active* and *Buffered* is checked on **EndFiring** events. With any inconsistency, the message's valid bit is inverted; if the message is in *Active* a cancellation message is sent and the processor is considered to have *faulted*.

7. Synchronize, and determine whether any processor faulted. If not, terminate (if appropriate) or set $s_{i+1} = w(s_i)$, $i = i + 1$, *Active*=*Received*=$\emptyset$, remove invalid messages from *Buffered*, and goto step 2 .

8. (Faulty window processing) Process cancellation messages by inverting valid bits on cancelled messages in *Received*.

9. Synchronize. Any processor receiving a cancellation message recovers its checkpointed state, and goes to step 5. All other processors go to step 7.

We have not yet implemented this algorithm. Issues to be examined are the cost of state-saving, the number of resimulations that are required on average to determine the correct behavior, and optimizations that permit a processor to realize it is insensitive to a message cancellation. If its costs turn out to be low, then synchronous relaxation is an attractive method for exploiting more parallelism than our strictly conservative method.

4 Automated Mapping

Given that the simulation has already been divided into LPs, our approach to the mapping problem has two components. First, at load-time, the LPs must be assigned to processors without knowing either the distribution or intensity of workload. This is accomplished using a static mapping algorithm, operating on topological estimates of workload. Secondly, at run-time, the program monitors the simulation activity of each LP. Based on these measurements, the program periodically decides whether to redistribute the LPs. The decision is based on projected finishing times under the present mapping versus a proposed mapping (computed using the static mapping algorithm applied to the measured workload). In this section we develop the static mapping algorithm and analyze some of its properties. Then we describe the dynamic remapping decision policy that governs when remapping occurs, and its implementation. Performance data presented in Section §5 is taken from runs managed by these techniques.

4.1 Static Mapping

Suppose that the TPN has been partitioned into a set of n LPs. We need to assign an execution weight e_i to each LP i, however, we have no idea of what its computational requirements will be. As an initial guess we could set e_i equal to the number of places and transitions in LP_i. Similarly, for every communicating pair i and j we assign an estimated communication cost; the rate of actual communication between i and j is unknown, as an initial guess we take the number of arcs crossing between the two. Now consider the mapping problem, given these costs. This weighted graph model has been studied in many contexts, assuming many different objective functions. The cost of parallel processing is best captured if we seek to minimize the load on the most heavily loaded processor, where the cost of a processor's load is defined to be the sum of the execution weights of its LPs, plus the sum of the costs of its LP's edges that are "cut" by the mapping (i.e., edges whose LPs lie on different processors). Formally, if e_i is the execution weight of LP i, $W(i,j)$ is the communication weight between LPs i and j, and $R_i(m)$ is the set of vertices assigned to processor i under mapping m, then the bottleneck cost is

$$\max_i \{ \sum_{j \in R_i(m)} e_j + \sum_{j \in R_i(m), k \neq R_i(m)} W(i,j) \}.$$

Unlike many other objective functions in the mapping literature, this one explicitly considers parallelism in both computation and communication. Fast algorithms for finding optimal mappings with respect to this function are known when the LPs are arranged in a linear order, and the mapping satisfies the contiguity constraint [3, 10, 26, 5]. This means that the workload assigned to a processor must be a contiguous subchain of LPs in the linear order. If we are to use these techniques we must rationally order the LPs, attempting to force the highest rates of communication to be between co-resident or nearby LPs. We then apply a fast mapping algorithm. Since the mapping algorithms themselves are not new, we focus on the problem of linearization. Following this we prove that the algorithm finds optimal mappings on balanced ring and hypercube graphs, and we explain why many different linearizations enable the chain mapping algorithm to find the optimal solution.

4.1.1 Linearization

We seek an ordering that concentrates highly communicating LPs close together everywhere throughout the ordering, because these have a much better chance of being co-resident. It is useful then to take a global rather than incremental view when ordering. Furthermore, it seems that even the simplest ways to quantify an ordering create a computationally intractable optimization problem. For example, the problem of maximizing the sum of weights between adjacent (in the ordering) LPs is a variation of the famous traveling salesman problem, which is is NP-complete. We desire even stronger conditions on our linearization, in keeping with its eventual usage. The linearized workload is to be partitioned into contiguous subchains, so that communication between LPs in the same subchain is essentially free. To measure this, consider an ordering π and the partitioning

of LPs under π into contiguous subchains of equal length 2^j (excepting the last one, which may be shorter). Let $S_j(\pi)$ denote the sum of weights on edges between LPs assigned to the same subchain, the sum being taken over all subchains. The larger $S_j(\pi)$ is, the less communication would occur between processors if the chain were partitioned as assumed. An ordering π that maximizes $S_j(\pi)$ for all $j = 1, 2, \ldots, \lceil \log |G| \rceil$ captures our desire that the more closely LPs communicate, the closer they are in the ordering.

We have developed a greedy approach based on a maximal weight matching algorithm [35]. We call this the *match/merge* algorithm. Given an undirected graph (G, E) with n vertices and m edges with nonnegative edge weights, a matching is a subset $M \subseteq E$ constrained so that no two edges in M share a vertex. A *maximal weight matching* maximizes the sum of edge weights possible in a matching. Sophisticated algorithms determine a maximal matching in $O(nm \log m)$ time [35]. If we weight the edges of an LP communication graph with estimates of communication volume per unit simulation time, then a maximal weight matching will find a good way of *simultaneously* pairing together LPs with high communication costs.

Our algorithm will ensure that LPs paired under the matching will be adjacent in the ordering, consequently any ordering π it discovers will always maximize $S_1(\pi)$. Of course, a single application of a matching algorithm will not linearize the LPs. In preparation for another matching step we reduce the graph size by merging paired LPs, into *super-LPs* (and any LP not paired in the matching is also considered to be a super-LP). Edges between super-LPs are defined naturally: an edge between super-LPs A and B exists if A contains an LP a and B contains an LP b such that a and b share an edge in the original LP graph. The weight of an edge in the super-LP graph is the sum of all edge weights of edges it represents, i.e., the total communication volume between LPs in A and LPs in B. Following this reduction, we apply the same matching algorithm to the super-LP graph. If super-LPs A and B are paired, then in our ordering the LPs represented by A and B will be adjacent in the sense that no LP from a super-LP other than A and B can lie between any two LPs in the concatenation (A, B). Following a pairing of super-LPs we can reduce the graph as before, and continue the process until the entire graph has been reduced to a single super-LP. The object at each step j of the algorithm then is to maximize $S_j(\pi)$, given the existing grouping into sets of size 2^{j-1}. The algorithm is described below.

1. Initialize n = number of LPs, index LPs from 0 to $n-1$. Initialize edge weights $W(A, B)$ for all LPs A, B. ($W(A, B) = 0$ if A and B do not communicate).

2. Find maximal weight matching M.

3. Order the super-LPs: for every $\{A, B\} \in M$, if index$(A) <$ index(B) then $C = (A, B)$ else $C = (B, A)$; index$(C) = \min\{\text{index}(A), \text{index}(B)\}$.

4. Renumber the merged super-LPs to maintain their represent ordering, but to range over $[0, \lfloor \frac{n}{2} \rfloor - 1]$.

5. If some LP Z is not matched under M, then index$(Z) = \lfloor \frac{n}{2} \rfloor$.

6. For all (A,B), (C,D) matched above,

$$W((A,B),(C,D)) = W(A,C) + W(A,D) + W(B,C) + W(B,D).$$

7. If n is odd then $n = \lfloor \frac{n}{2} \rfloor + 1$, else $n = \lfloor \frac{n}{2} \rfloor$. If $n > 1$ goto step 2.

The ordering of merged two super-LPs ((A,B) vs. (B,A)) is specified in terms of inherited index numbers. This is somewhat arbitrary. While values of $S_j(\pi)$ are insensitive to this choice, other objectives (such as distributing workload evenly along the chain) are not. The match/merge process is depicted naturally by a binary tree whose leaf vertices represent original LPs, and where parent-child relationships reflect merging decisions. Given the merging decisions, each possible linearization is uniquely determined by a set of decisions that order each interior vertex's children. As there are $n/2$ interior vertices, then there are $2^{n/2}$ different linearizations derivable from a given set of match decisions. If the decision is made that child A precedes child B, the effect is that all LPs descended from A will precede all LPs descended from B in the ordering. It might be useful to delay ordering decisions until the tree is constructed, and then choose orderings to spread out the workload as well as to better localize the communications. We have not yet explored this problem, but feel it is worthy of future attention.

We can bound the deviation from optimal of the linearizations produced by this method. Let π^h be a linearization produced by our heuristic, and for every j let π_j^{opt} be a linearization that maximizes $S_j(\pi)$ over all linearizations π.

Lemma 1 *For all* $j = 1, 2, \ldots, \lceil \log |G| \rceil$,

$$\frac{S_j(\pi_j^{opt})}{S_j(\pi^h)} \leq (2^j - 1)\frac{S_1(\pi^h)}{S_j(\pi^h)}.$$

Proof: Let S_1 be the set of first 2^j vertices under π_j^{opt}, S_2 be the second set, and so on. The vertices in every set S_i have up to $2^j(2^j-1)/2$ number of edges between them. These edges can be partitioned into 2^{j-1} sets $E_{i,k} = \{\{i, (i+k) \bmod 2^j\} \mid i = 0, 1, \ldots, 2^j - 1\}$, for $k = 1, 2, \ldots, 2^{j-1}$. Note that each $E_{i,k}$ is a matching on S_i. Now for every k, $\cup_i E_{i,k}$ is a matching on the unordered graph (G,E). Since $S_1(\pi^h)$ maximizes the sum of weights of a matching on (G,E), we have

$$S_j(\pi_j^{opt}) \leq (2^j - 1)S_1(\pi^h).$$

The result is obtained dividing through by $S_j(\pi^h)$. ∎

Note that since $S_j(\pi^h) \geq S_1(\pi^h)$ for all j, we get a loose "pre-computation" bound of $2^j - 1$. This bound will be sharpened given measured values of $S_1(\pi^h)$ and $S_j(\pi^h)$.

Another feature of a match/merge algorithm linearization is that the sum of edge weights between adjacent LPs is not less than half of optimal.

Lemma 2 *Let* ω^{opt} *be the maximum possible sum of edge weights between adjacent LPs in any linear ordering, and let* ω^h *be the sum of such weights under a linearization produced by the match/merge algorithm. Then* $\omega^{opt}/2 \leq \omega^h$.

Proof: Let π^{opt} be a linearization that maximizes the sum of weights between adjacent LPs, and renumber the LPs with respect to π^{opt}. Let S_{even} be the set of edges of the form $(i, i+1)$, for i even, and let S_{odd} be the set of edges of the form $(i, i+1)$ for i odd. Both S_{even} and S_{odd} are matchings, hence the sum of edges in either is no greater than $S_1(\pi^h)$. This gives

$$\omega^{opt} \leq 2S_1(\pi^h) \leq 2\omega^h,$$

from which the result follows. ∎

This result is essentially the same as a similar one for a TSP heuristic based on a matching step [18].

To accelerate solution time (possibly at the expense of solution quality) we normally use a $O(n)$-time approximation to the maximal weight matching step, called a "stable" matching. This is one in which it is not possible to find matched pairs (A, B), (C, D) such that

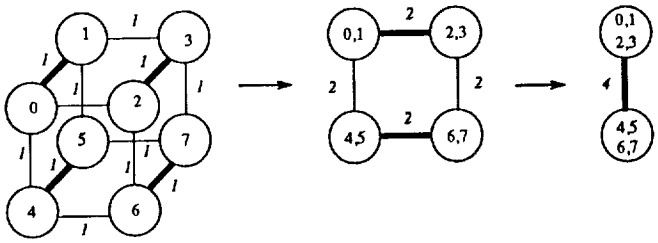
$$\max\{W(A,C) + W(B,D), W(B,C) + W(A,D)\} > W(A,B) + W(C,D).$$

Such an algorithm is obtained from a modification to the stable marriage problem [31] for bipartite graphs. The sense of the original stable marriage solution is to loop over all "males", each one attempts to become engaged by proposing to the "females" in decreasing order of preference. If a suitor finds a previously engaged debutante such that the debutante prefers the new suitor to her engaged, the old engagement is broken and a new one forged, and the jilted suitor is left to pick up and continue his search for a mate.

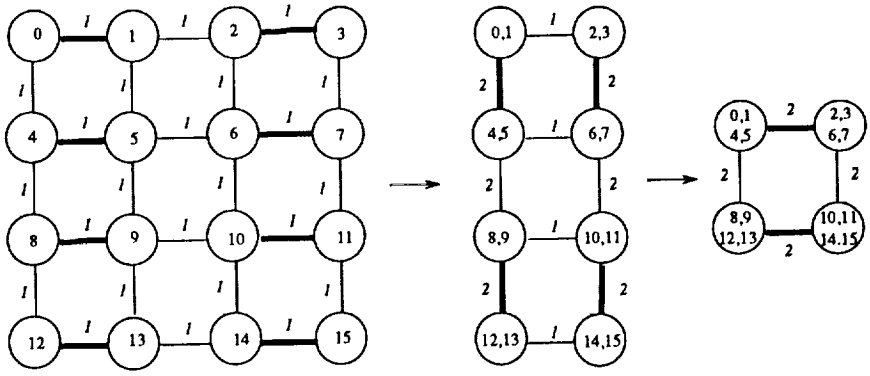
Restated in our context, every LP orders all other LPs with which it communicates. Higher communication implies higher preference. We cannot immediately apply the stable marriage algorithm, as we lack distinct sexes. A minor modification to the algorithm has the effect of selecting sexes: once an LP becomes engaged playing the role of a debutante, it is not later considered to be a suitor. This effectively separates the LPs into two equal sized groups; the matching found will be stable with respect to the sex roles discovered during the process.

The effects of match/merge using stable matching on a 3-dimensional hypercube and on a 2-dimensional mesh are illustrated in Figure 1. Ties between equal weighted edges are resolved in favor of LPs with lower index numbers (this is an important aspect of the match process when matching balanced graphs). We see that the hypercube with equal weight edges collapses along dimension lines, and that the mergings in the mesh alternate between dimensions. Linearizations based on these processes are clearly dealing with the global structure of the graph rationally.

Petri-net models of parallel architectures exhibit high connectivity locally, for instance, reflecting the interconnection pattern of a modeled parallel architectures. This feature has an impact on the algorithmic cost of linearization. If each of n LPs communicates with every other LP then there are $O(n^2)$ distinct inter-LP communication costs to calculate, and the computation of preferences requires $\Omega(n^2 \log n)$ time. However, these costs drop to $O(nB)$ and $\Omega(nB \log B)$ if an LP communicates with no more than B others. This makes a real difference when n is large, and $B \ll n$. It is also true for our CM-1 router example.



Match/Merging a Hypercube with unit edge weights



Match/Merging a Mesh with unit edge weights

Figure 1: Behavior of match/merge algorithm on a hypercube, and mesh

The cost of a matching step is dominated by the cost of computing and sorting inter-LP communication costs. The first step exacts an $O(nB \log B)$ cost. At the second step the number of super-LPs involved is halved, but in the worse case the number of connections a super-LP has doubles. This gives the second step a cost of $O(nB \log(2B))$. In general the i^{th} step costs $O(nB \log(2^{i-1}B))$; the cost sum over all $\log n$ steps is

$$\begin{aligned} O\left(\sum_{i=1}^{\log n} nB \log(2^{i-1}B)\right) &= O\left(nB \sum_{i=1}^{\log n} (\log B + \log 2^{i-1})\right) \\ &= O\left(nB(\log B)\log n + nB \log^2 n\right) \\ &= O\left(nB \log^2 n\right). \end{aligned}$$

One of the attractions of the match/merge algorithm is that on certain graphs it is optimal in the sense that the linearization it finds simultaneously maximizes $S_j(\pi)$ for all j. We will specifically argue for its optimality when applied to rings and to hypercubes. Then we'll examine an asymptotic bound on the deviation of the algorithm from optimality on a multidimensional mesh. In all of these cases the communication topology is very regular, and we assume unit edge weight costs. This

situation corresponds to an initial mapping of a homogeneous Petri net model of such architectures, prior to run-time measurement of execution and communication costs. This is still an important problem, because even with measured costs the model may well be uniformly weighted. This in fact was an unexpected consequence of our CM-1 router example.

LP enumeration has a definite affect on the matches made, and a little care is required for our optimality results to hold. For the specific cases we consider we suppose the LPs to be enumerated in a "natural" way. We presume a ring is enumerated so that adjacent vertices in the enumeration share a communication edge; we presume that vertices in a $2^k \times 2^k \times \cdots 2^k$ torus are enumerated in row-major order, just as they would be in an multi-dimensional array; we presume the usual enumeration of a hypercube where vertex i and j share an edge if and only if the Hamming distance between i and j is exactly one.

Lemma 3 *Let (G, E) be a ring, enumerated naturally, with unit edge weights, and $|G| = 2^k$. Then for all $j = 1, 2, \cdots, k$, any linear ordering π produced by the match/merge algorithm maximizes $S_j(\pi)$.*

Proof: For any integer j, it is obvious that the partition into 2^{k-j} pieces maximizing the number of edges between vertices in a common partition element is obtained by grouping the first 2^j vertices together, then the next 2^j, and so on. This is precisely the grouping defined by the match/merge algorithm. ∎

Lemma 4 *Let (G, E) be a hypercube of dimension k, suppose that G is enumerated naturally, and that all edges have unit weight. Then for all $j = 1, 2, \cdots, k$, any linear ordering π produced by the match/merge algorithm maximizes $S_j(\pi)$.*

Proof: We first induct on x to prove that the number of edges between members of any subset of x vertices is no greater than $(x \log x)/2$. The base case of $x = 1$ is trivially satisfied. Suppose then that the claim is true for any subset of size $x - 1$ or smaller, and choose any subset A with x vertices. Split A evenly into two subsets A_1 and A_2. The number of edges between vertices in A is the sum of the edges on A_1 plus the edges on A_2 plus the edges between them. There are at most $\lfloor x/2 \rfloor$ edges between them, and by the induction hypothesis the sum of edges on A_1 and on A_2 is no more than $(x \log(x/2))/2$. Therefore the number of edges on A is no more than

$$\lfloor x/2 \rfloor + \frac{x \log(x/2)}{2} \leq \frac{x \log(x)}{2},$$

which completes the induction. Now observe that when $x = 2^i$ the bound is met, and is met by sets that themselves form hypercubes (which have $x2^{x-1}$ edges contained within them). Now at every step i the match/merge algorithm merges hypercubes of dimension $i - 1$ into hypercubes of dimension i (a consequence of G being ordered naturally). Thus $S_j(\pi)$ is maximized for each j. ∎

Finally, consider a d-dimensional mesh, where G may be placed in one-to-one correspondence with integer-vector elements of $[1,2^k]\times[1,2^k]\times\cdots[1,2^k]$, and edges exist between nearest neighbors in each dimension. Assuming a natural ordering and unit weight edges, the match/merge algorithm cycles through each dimension, i.e., at the j^{th} step it merges all vertices that are neighbors in dimension $j \bmod d$. The super-LPs it aggregates are themselves nearly cubic submeshes. The problem of partitioning such a grid optimally has been studied in the context of meshes used for numerical problems [28]. These studies look at the ratio of computation to communication costs, where computation is measured as the number of mesh points in a partition element, and the communication cost is measured as a function of mesh edges between partition elements. For our purposes we consider the communication cost to be simply the number of such edges. In this context the problem of maximizing $S_j(\pi)$ is the problem of partitioning the mesh into 2^{d-j} pieces so as to maximize the computation/communication ratio of a partition element. It is known that cubic partitions are not optimal, being bested for instance by hexagonal partitions[28]. We can bound however the asymptotic deviation of cubic partitions (and hence the asymptotic deviation of our $S_j(\pi)$ from optimal) by considering the continuum limits. For a given volume V in Euclidean d-space, the topology that maximizes the volume to surface ratio is a sphere. Surface area here directly corresponds to numbers of cut edges, if we think of the sphere as enclosing many many mesh points, and we count the edges cut by the sphere's outer shell. Let $r_d(V)$ be the radius giving rise to volume V for a sphere in d-space. The ratio of volume to area is $r_d(V)/d$ (a fact established from straightforward principles of calculus). Suppose then that $V = s^d$, the volume of a cube with side s in d-space. The ratio of volume to surface of the cube is $s^d/(2ds^{d-1})$. It follows then that the ratio of the cube's surface to the sphere's surface given volume s^d is just $2r_d(s^d)/s$. As d increases, $r_d(s^d)/s$ decreases, hence we may use $r(s^2)/s = \sqrt{\pi}$ as an upper bound for all $d \geq 2$. Consequently, the asymptotic ratio between the number of cut edges in a cube to that cut under the optimal partition (which will maximize $S_j(\pi)$ and thereby minimize "surface") is no more than $2\sqrt{\pi}$.

4.1.2 Chain Mappings

Suppose that some linearization of the LPs is given. The most general formulation of the remaining mapping problem allows any two LPs in the linear order to have non-zero communication costs. A dynamic programming programming formulation solves the problem in $O(Pn^2)$ time, P being the number of processors. To see this, let $C(j,p)$ be the optimal bottleneck cost achievable mapping LPs 1 through j onto p processors. Then the principle of optimality asserts that

$$C(j,p) = \min_{i<j}\{\max\{C(i,p-1), \sum_{k=i+1}^{j} e_k + \sum_{k=i+1}^{j}\sum_{m\leq i, m>j} W(k,m)\}\}.$$

A key thing to remember is that this solution permits non-adjacent LPs to have non-zero communication costs. Previous treatments of the chain mapping problem have restricted their attention to the alternate case. Not surprisingly, under the more constrained assumption the algorithms have lower complexity.

One of the interesting aspects of the match/merge/map approach is that we can prove there are instances of the mapping problem where the approach will find the optimal mapping. First we show that match/merge/map finds the optimal solution for rings and hypercubes with unit communication costs on each edge and common execution costs for each LP. This may seem weak, but is stronger than it looks at first glance owing to the tension between load balance and communication costs. Furthermore, it is not an assurance that is given by other mapping heuristics. The second result is entirely general. We show that that any optimal mapping can be embedded in a linearization, and that there are a great many different equivalent linearizations upon which the chain mapping algorithm will discover a solution with optimal cost. This result gives us some measure of the resiliency of restricting our attention to chain mappings.

The first conclusion is obvious.

Lemma 5 *Let (G, E) be a ring, enumerated naturally, with unit edge weights. Suppose every vertex has common weight w. Then for any power-of-two number of processors P, the match/merge/map algorithm minimizes the bottleneck over all possible partitions of the ring into nonempty P sets.*

Proof: Under any mapping, every processor has a communication cost of at least two. The linearization produced by the match/merge algorithm gives every processor a communication cost of exactly two. The chain mapping algorithm will map no more than $\lceil |G|/P \rceil$ LPs to any processor, yielding a bottleneck cost of $2 + w\lceil |G|/P \rceil$, which is optimal. ∎

The second conclusion shows that in balanced hypercubes, for moderate values of w it is optimal to assign equal sized hypercubes of smaller dimension to each processor—as does match/merge/map.

Lemma 6 *Let (G, E) be a hypercube, enumerated naturally, with unit edge weights, and $|G| = 2^k$. Suppose every vertex has common weight $w \geq 1/(2 \ln 2)$. Then for any power-of-two number of processors $2^j \leq 2^k$, the match/merge/map algorithm minimizes the bottleneck over all possible partitions of the hypercube into up to 2^j pieces .*

Proof: Consider a processor assigned any x LPs. The proof of lemma 4 shows that the sum of edges between LPs on that processor is no greater than $(x/2)\log x$; hence there are at least $(k - x\log x)/2$ edges to LPs on other processors. The function

$$f(x) = wx + \frac{kx - x\log x}{2}$$

is thus a lower bound on the cost of assigning x LPs to a processor. Note that the bound is achieved if the set forms a hypercube. Considering x as continuous, we have

$$f'(x) = w + \frac{k - \log x}{2} - \frac{1}{2\ln 2}.$$

Note that $\log x$ is maximized when equal to k, hence f is increasing over $x \in [0, 2^k]$ if $w \geq 1/(2\ln 2)$. If $x_1, x_2, \ldots x_{2^j}$ are workload assignments ($x_i \geq 0$, $\sum_{i=1}^{2^j} x_i = 2^k$ then the function

$$g(x_1, \ldots, x_{2^j}) = \max\{f(x_1), f(x_2), \ldots, f(x_{2^j})\}$$

is a lower bound on the bottleneck cost of the assignment. Since f is increasing, g is minimized when the maximum x_i is as small as possible—that is, when the x_i's are identically 2^{k-j}. This situation is achieved when the LPs are partitioned into 2^j hypercubes, furthermore the value of g then is also exactly the bottleneck. If G is enumerated naturally, the match/merge/map algorithm will produce this assignment. ∎

Other situations where the match/merge/map approach finds optimal solutions occur as a result of the definition of the bottleneck cost. Any solution that minimizes the bottleneck can be embedded in a linearization. For example, given the optimal mapping we can renumber the LPs assigned to processor 1 starting at 1, then carry over the enumeration to LPs assigned to processor 2, and so on. However, a large number of linearizations are equivalent in the sense that the chain mapping algorithm will find the optimal bottleneck on them. For example, any permutation of the processor ordering does not affect the bottleneck cost, and does not confuse the chain mapping algorithm. Likewise, within the LPs assigned to a processor there is an insensitivity to their ordering within the processor. The net effect is that given an optimal solution and an associated linearization π^{opt}, there are a number of permutations of π^{opt} that will not affect the sets of LPs that are co-resident. Given any one of these linearizations the chain mapping algorithm with discover the optimal bottleneck.

Lemma 7 *For any mapping problem involving m LPs and P processors and minimized bottleneck cost b, there are at least $P! \times \Gamma(m/P)^P$ different linearizations upon which the chain mapping algorithm will discover a solution with cost b.*

Proof: Suppose that a solution minimizing the bottleneck value b assigns n_i LPs to processor i, for $i = 1, 2, \ldots, P$. From the discussion above there are at least $P! \times \prod_{i=1}^{P} n_i!$ different linearizations for which the chain mapping algorithm will produce a solution with bottleneck cost b. In the continuous domain, $\Gamma(n_i) = n_i!$, and for fixed $\sum_{i=1}^{P} n_i = m$ the product $\prod_{i=1}^{P} \Gamma(n_i)$ is minimized when $n_1 = n_2 = \ldots n_P = m/P$. ∎

4.2 Dynamic Remapping

Our approach to dynamic remapping has essentially been laid out before, in [27], with an emphasis on physical computations that exhibit distinct phases. The issue there is to determine with sufficient confidence that a phase change has occurred and that performance will benefit from remapping. The general approach is to periodically consult an "oracle" that judges whether it is worthwhile to remap now. The oracle's decision is not immediately acted upon though, it is used to update (via Bayes Theorem) a *gain probability* that performance will improve by remapping now. The optimal decision policy was shown to be a threshold policy—if the gain probability is larger than some step-specific threshold, then one ought to remap. As computation of the optimal decision thresholds proved to be impractical, a heuristic was proposed to use a constant, high, threshold.

We apply this work to the present context, as follows. Periodically, just prior to the beginning of a window's processing the processors all coordinate to make a remapping decision. The decision

is based on measurements of the average processing cost undergone so far by every LP. As the simulation runs, a processor keeps track of the number of events executed so far on behalf of each LP. The processor also keeps track of the total time spent so far processing events, by measuring the time spent by a routine which in one call processes all the events done by a processor in a window. With these figures we compute the average time spent by each LP processing events in a window; the average may be exponentially decayed to allow sensitivity to time-varying averages. The averages become the LP weights for the static mapping algorithm. It makes a great deal of sense to balance based on these per-window averages, since windows are separated by barrier synchronizations. Furthermore, given these averages, a prospective mapping, and knowledge of the simulation's termination time (in simulation time) we may estimate the remaining time required to complete the simulation under the assumed mapping. Our approach then is to have an oracle routine compute the best mapping given the present LP workload averages, then predict the expected time to remap (with an estimated remapping cost of 1 second) and finish the computation under the new mapping. The oracle similarly predicts the expected finishing time if one does not remap. The oracle recommends remapping if its projections suggest a performance improvement of at least 10%. This judgement is used to update the gain probability. The purpose of the 10% padding is to protect from underestimating the remapping cost (which isn't known until it is observed). Since we have observed that the initial mapping can be truly inferior, we modified the heuristic so that a remapping is performed automatically if the oracle judges the new mapping to be twice or more faster than the old. In our experiments we have seen remapping triggered both by the gain probability crossing the threshold (which requires 2-3 consecutive positive oracle judgements), and by the twice-as-good rule. The remapping logic is not disabled after the initial remapping; if the initial remapping decision turns out to be very wrong it is still be possible to correct it. Similarly, if the workload has a time-varying average, then the decayed sample averages can reflect this, and trigger a remapping.

Our implementation of this policy deserves comment, as it would be easy to implement the logic inefficiently. The basic idea is to provide every processor with an estimate of every LPs workload, and then have every processor execute exactly the same code and make exactly the same remapping decision as all the others. We attempt to distribute workload information with relatively little communication, as follows. In a first step we compute the maximum and minimum LP loads throughout the system, using a software combining tree that operates on vectors (e.g., a global reduction min on vectors). Such reductions are supported by fast library routines on Intel multicomputers. Next, every processor discretizes the range of LP workloads into 16 levels. Then, for each of its LPs, a processor generates a 4-bit code describing which of those levels best describes the LPs load. These 4-bit codes are inserted into an array, initially empty, of codes for all LPs. The processors engage in a global bitwise-OR reduction on this table, which serves to distribute the code information to every processor. From these codes the processors can now reconstruct the same approximate workload levels as any other processor, and execute the remapping logic based on these estimates. Every processor computes which LPs it must shed, and which ones it will receive. The actual disengagement of an LP from one processor and integration into another involves some work

related to bundling and unbundling state information, and keeping the processors data structures up to date.

We have chosen to ignore communication costs in the mapping step, for two reasons. First, the number of different communication values is quadratic in the number of LPs, which implies a great deal of information to gather and distribute at run-time. Intuition suggests that if the linear ordering is successful in keeping closely communicating LPs together, then ignoring the communication costs while mapping should not grossly affect performance. This does beg the issue of recovering from a bad linearization.

Three areas of the scheme above bear further investigation. The bit-vector approach to distributing load information is efficient for for small-to-medium numbers of LPs, after which the communication cost can be overly high, requiring a different method. and so a different approach for computing the tentative Secondly, certain parameters may need dynamic adjustment, particularly the frequency of computing tentative mappings, and the decay parameter for workload averaging. Thirdly, we ought to investigate dynamic re-linearization.

5 Empirical Study

Our empirical study considers TPN models of a a slotted ring network, and of the Connection Machine CM-1 global routing network. Our models to not attempt to accurately capture all aspects of behavior; instead they are intended to be representative of large TPN problems to which one might apply parallel processing.

The CM-1 network model is based on the description in [1]. It captures the dimension-by-dimension structure of message-passing, the effects of limited buffer space, and the interaction between a router and the mesh of processors that directly access it. Every node of the global network serves 16 PEs; a PE signals its decision to communication (made randomly) by placing a token in a specified location. For each message, a dimension is chosen and the message is enqueued to be sent across that dimension. When a message arrives at a new PE, a random decision is made to either absorb the message (modeling its terminal arrival), or to send it through another dimension, chosen uniformly at random among all dimensions higher than the one through which it came. The model explicit mimics the petit and grand cycle nature of the CM-1, and explicitly mimics handshaking that ensures a buffer is available for a message before it is sent. The dimension d of the hypercube parameterizes this model. A model with 6 dimensions has over 10,000 places and 10,000 transitions. A model with 8 dimensions has over 100,000 places and transitions.

The slotted ring model is comprised of some N LPs that model fine-grained workload, we call these *workload generators*. A workload generator is basically a loop within which a set of tokens circulate. Every pass through the loop (five events) the workload generator randomly decides whether to communicate over the ring. The ring itself is an LP, hence the communication topology of the system is a tree with one root (the ring) and N leaves. The simulated communication is non-blocking and serves simply to generate some simulation workload (a round-trip for a message) for the ring LP. We control workload intensity by varying the firing time on loop transitions. The smaller

the firing time, the more simulation work per unit simulation time is required. This model clearly demonstrates the need for dynamic remapping, because reasonable workload estimates derived from the topology alone fail miserably to balance the load. A model with 64 workload LPs and one ring LP has 4672 places and an equal number of transitions.

The timing delays in both models are based on realistic disparities between computation and communication times. This has a definite impact on the synchronization protocol. For example, certain "slow" transition firings in the CM-1 network model are two orders of magnitude larger than the smallest delays on firings that cross processor boundaries. There is a very sizable lag (in simulation time) between the last "fast" transition to fire, and the firing of the slow transition. It would be disastrous to simply advance time by the minimum boundary firing time amount each window, for many windows would contain no events. However, the technique of adding the least on-processor event time-stamp to the minimum boundary firing time effectively skips over these periods.

The simulations were conducted on the YAWNS (Yet Another Windowing Network Simulator) parallel simulation testbed [19, 25], implemented on the Intel family of multiprocessors. We present data from runs executed on the Intel iPSC/860, and upon the Intel Touchstone Delta [14], a large-scale multiprocessor also based on Intel's i860 CPU. The time spent in the merge/match/mapping algorithm was dominated by the IO time to read the network description, and so proved to be inconsequential.

The CM-1 routing network example defines an LP naturally as the submodel associated with one router node. The problem communication topologies thus forms a hypercube. Since all LPs are structurally identical, any topological measure of workload will assign the same workload to each LP, and the merge/match/map algorithm maps it optimally under the assumptions of uniform communication and execution costs. Moreover, under the homogeneous model assumptions, the average execution cost for every LP was identical. No long term load imbalances could be expected to develop, so that dynamic remapping should probably not be used.

Figure 2 plots the measured performance of the CM-1 model on sixteen processors of the Intel iPSC/860, as a function of the hypercube dimension ($d = 6, 7, 8, 9$). The various random decisions were given parameters to cause the greatest amount of interprocessor communication. Each problem size was executed long enough to simulate over ten million events, and was simulated both with remapping logic enabled, and disabled. The dynamic remapping mechanism never caused the initial mapping to be abandoned, even though at some individual remapping assessments it appeared (temporarily) that some gain might be achieved by rearrangement. This illustrates the essential safety offered using the Bayesian filter.

The left vertical axis demarks the aggregate average number of events executed per second (in units of a thousand). The right vertical axis delineates the corresponding speedup, measured using an optimized serial code that employs the same Petri net event processing logic, but uses a splay-tree priority list to manage events. All speedup measurements are computed using the measured serial rate on a six dimensional problem, as the larger models would not fit in one node's memory.

The overhead of gathering workloads and projecting remapped performance can be seen as

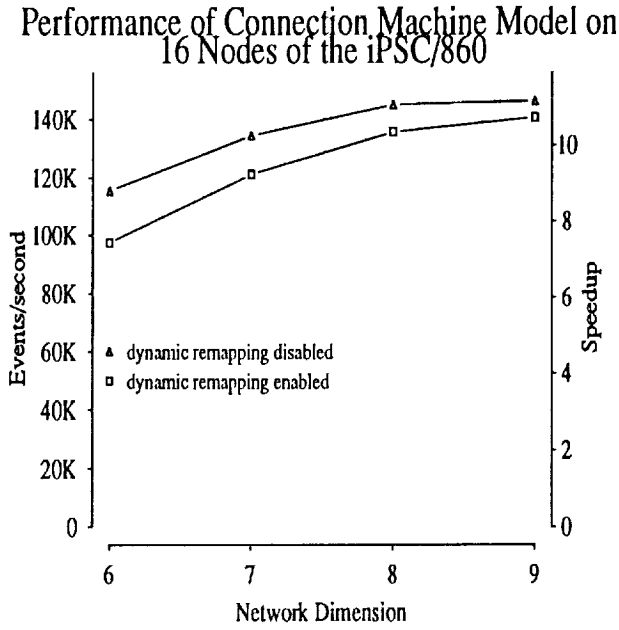


Figure 2: Performance on Connection Machine global router example

the gap between the two performance curves. On this problem the difference is less than 10%, a difference that is increasingly amortized as the problem size grows. With increasing problem size performance gets better, but it is clear that if the growth trend continues, by dimension 9 the event rate is close to its maximal level. The fact that this occurs at a speedup less than 12 is due to the cost of communication on the iPSC/860, which is quite high relative to the speed of the CPU. These same speedups on the more balanced Intel iPSC/2 are nearly 20% better (but the iPSC/2's CPU is a factor of 7 slower on this problem!).

We have also investigated our synchronization algorithm on various TPN models of mesh-based architectures; these results are reported in [20]. Like the CM-1 model, these are essentially self-balancing (at the time of that paper we had not yet developed the mapping and remapping methods). This study also shows increasing performance as the problem size grows, but also shows the dependence of good performance on a favorable computation to communication ratio. The cost of communication on the Intel iPSC/2, iPSC/860, and Delta architectures we've used is high enough to require substantial computation on average between communications.

We use the slotted ring model to better explore the benefits of dynamic remapping. As a test case for unbalanced workload, we created a model with 64 workload LPs where the first 6 and last

47 LPs generate events at a rate of 50 events per unit simulation time, while the remaining 17 LPs generate 500 events per unit simulation time. This particular assignment of workload stresses the static mapping algorithm, as adjacent heavy workloads are harder to distribute under its linear ordering constraints. However, the initial assignment estimates workloads based solely on topology, and is unable to distinguish between heavy and light workloads. Furthermore, the ring LP has many more places and transitions than a workload LP, but ends up having nowhere near the same event intensity. As a consequence we can usually expect the initial mapping to be very bad. Finally, the unbalanced workload model is simple enough to compute an upper bound on the speedup possible, by assuming the LPs are distributed optimally (an easy calculation by hand), communication costs are free, and the initial mapping is perfect.

Figure 3 plots the results of simulating this model on 16, 32, and 64 processors of the Intel Touchstone Delta. All runs generated approximately 10 million events. The vertical axes are as before, this time with the event execution rate expressed in millions of events per second. We plot three sets of data associated with "bad balance", the unbalanced workload described above. For the purposes of comparison we also plot data associated with a "good balance" model where all LPs have the same weight (50 events per unit simulation time). In all these runs communication with the ring LP is infrequent. This allows us to isolate the effects of load imbalance from communication costs. We still do have inescapable communication costs due to synchronization, which occurs every unit of simulation unit.

On all runs where remapping was employed, remapping was chosen very shortly into the run, as it was quickly evident that a new mapping based on event count measurements was superior. Although theoretically possible, no subsequent remappings were performed. The necessity of dynamic remapping is clearly seen by examining performance when remapping is disabled. The jump in performance at 64 processors is a consequence of the mapper initially always using as many processors as are available. The worst that can happen (which did) is that the one processor assigned two LPs gets two workload LPs.

We also see that the dynamic remapping mechanism comes close to achieving the optimal performance possible, given the unbalanced workload. Performance of the balanced workload is nearly perfect for 16 and 32 processors; it falls away at 64 processors owing to the low number of events performed on each processor between synchronizations (50).

6 Summary

This paper studies the problem of automatically parallelizing the discrete-event simulation of large timed Petri-nets executing on parallel architectures. The methods we described have been implemented in a tool where one designs a Petri-net using a graphical tool, and then all remaining steps for parallelization are performed automatically.

We describe a synchronization algorithm and automated load balancing techniques, both static and dynamic. We present a new static mapping algorithm, and study its properties analytically. This algorithm is not restricted to TPN simulations, it applies to more general parallel compu-

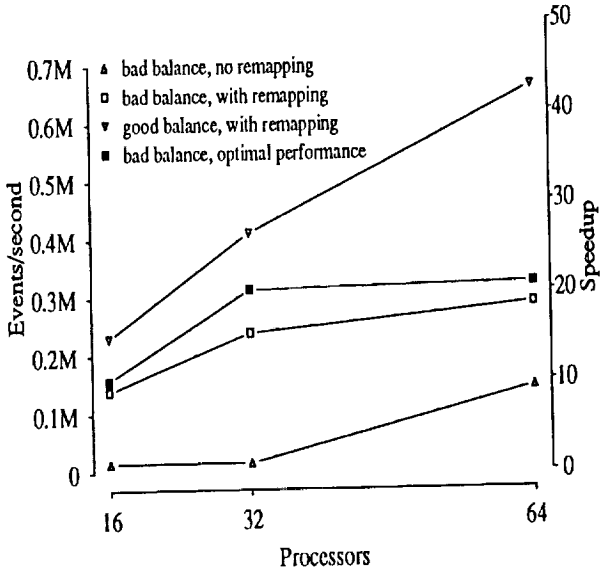


Figure 3: Performance on slotted ring network example

tations. We study the effectiveness of our methods on the performance of a simulation of the Connection Machine CM-1 routing network on 16 processors of an Intel iPSC/860, and on a simulation of a slotted-ring architecture that is executed on up to 64 processors of the Intel Touchstone Delta. Significant performance benefits are observed, and the effectiveness of (and need for) dynamic remapping clearly demonstrated.

Acknowledgements

We thank Subhas Roy for his programming contributions.

References

[1] G. S. Alamsi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989.

[2] F. Baccelli and M. Canales. Parallel simulation of stochastic petri nets using recurrence equations. *ACM Trans. on Modeling and Computer Simulation*, 3(1):20–41, January 1993.

[3] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

[4] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[5] H.-A. Choi and B. Narahari. Algorithms for mapping and partitioning chain structured parallel computations. In *Proceedings of the 1991 Int'l Conference on Parallel Processing*, St. Charles, Illinois, August 1991.

[6] S. Eick, A. Greenberg, B. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. In *Proceedings of the 1991 Workshop on Parallel and Distributed Simulation*, pages 151–162, Jan. 1991.

[7] G.A. Frank, D.L. Franke, and W.F. Ingogly. An architecture design and assessment system. *VLSI Design*, pages 30–38, August 1985.

[8] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[9] D.W. Glazer. *Load Balancing Parallel Discrete-Event Simulations.* PhD thesis, McGill University, May 1992.

[10] M.A. Iqbal and S.H. Bokhari. Efficient algorithms for a class of partitioning problems. Technical Report 90-49, ICASE, July 1990.

[11] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, July 1985.

[12] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLorento, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger. The Time Warp Operating System. *11th Symposium on Operating Systems Principles*, 21(5):77–93, November 1987.

[13] D. Kumar and S. Harous. An approach towards distributed simulation of timed petri nets. In *Proceedings of the 1990 Winter Simulation Conference*, pages 428–435, New Orleans, LA., December 1990.

[14] Sigurd L. Lillevik. The Touchstone 30 gigaflop DELTA prototype. In *Distributed Memory Computer Conference 91*, pages 671–677. IEEEPRESS, April 1991.

[15] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[16] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[17] B. Nandy and W. Loucks. An algorithm for partitioning and mapping conservative parallel simulation onto multicomputers. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 139–146. SCS Simulation Series, Jan. 1992.

[18] G. L. Newhauser and L.A. Wolsey. *Integer and Conbinatorial Optimization.* Wiley and Sons, New York, 1988.

[19] D. Nicol, C. Micheal, and P. Inouye. Efficient aggregaton of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

[20] D. Nicol and S. Roy. Parallel simulation of timed petri nets. In *Proceedings of the 1991 Winter Simulation Conference*, pages 574–583, Phoenix, Arizona, December 1991.

[21] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research.* To appear.

[22] D.M. Nicol. *The Automated Partitioning of Simulations for Parallel Execution.* PhD thesis, University of Virginia, August 1985.

[23] D.M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, September 1988.

[24] D.M. Nicol. Performance bounds on parallel self-initiating discrete event simulations. *ACM Trans. on Modeling and Computer Simulation*, 1(1):24–50, 1991.

[25] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[26] D.M. Nicol and D.R. O'Hallaron. Improved algorithms for mapping parallel and pipelined computations. *IEEE Trans. on Computers*, 40(3):295–306, 1991.

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

[28] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. on Computers*, C-36(7):845–858, July 1987.

[29] P.F. Reynolds, Jr. Comparative analyses of parallel simulation protocols. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., December 1989.

[30] L. A. Sanchis. Multiple-way network partitioning. *IEEE Trans. on Computers*, 38(1):62–81, January 1989.

[31] R. Sedgewick. *Algorithms.* Addison-Wesley, New York, 1988.

[32] H. Sellami and S. Yalammanchili. Efficient parallel simulation of marked graphs. In *Proceedings of the SCS Summer Simulation Conference*, pages 328–333, July 1992.

[33] H. Sellami and S. Yalammanchili. Consevative parallelsimulation of a class of multiprocessor simulation models. Technical Report TR-GIT/CSRL-93/02, Georgia Tech. School of Electricaaal and Computer Engineering, July 1993.

[34] T. Som. *Allocation of Processing Power in Optimistic Parallel Discrete-Event Simulation.* PhD thesis, Syracuse University, 1992.

[35] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Indusrial and Applied Mathematics, Philadelphia,PA, 1983.

[36] G. Thomas and J. Zahorjan. Parallel simulation of performance petri nets: Extending the domain of parallel simulation. In *Proceedings of the 1991 Winter Simulation Conference*, pages 564–573, Phoenix, Arizona, December 1991.

Reliability Analysis of Complex Models using SURE Bounds

David M. Nicol*
College of William and Mary
Williamsburg, VA 23187

Daniel L. Palumbo
NASA Langley Research Center
Hampton, VA 23668

February 9, 1993

Abstract

As computer and communications systems become more complex it becomes increasingly more difficult to analyze their hardware reliability, because simple models may fail to adequately capture subtle but important model features. This paper describes a number of ways we have addressed this problem for analyses based upon White's SURE theorem. We point out how reliability analysis based on SURE mathematics can be extracted from a general C language description of the model behavior, how it can attack very large problems by accepting recomputation in order to reduce memory useage, how such analysis can be parallelized both on multiprocessors and on networks of ordinary workstations, and observe excellent performance gains by doing so. We also discuss how the SURE theorem supports efficient Monte Carlo based estimation of reliability, and show the advantages of the method.

*This research was supported in part by NASA grants NAG-1-1060, NAG-1-1132, and NSF grant CCR-9201195.

1 Introduction

White's SURE theorem has laid the foundation for a number of reliability tools, including SURE [3] itself, ASSIST [2, 11], TOTAL, and PAWS. The latter three tools provide the user with a formal framework within which a model is described, then use the model description to explicitly build a semi-Markov state-space. The tool SURE is then applied, determining upper and lower bounds on the transient probability of the system entering a state reflecting system failure (i.e., a *death-state*) within a specified period of time. These tools have a large user base and have proven to be very useful in a wide range of contexts. For example, a survey conducted by NASA Langley found that the ASSIST/SURE toolset is used by United Technologies to model redundant engine control architectures, by Boeing to model fighter flight control systems, by Raytheon to model space-borne systems, by Rockwell-Collins to evaluate trade-offs in the reliability and safety of primary flight control architectures, and General Electric to model engines, engine controllers, locomotive engines, and satellite controllers. Industrial interest in SURE-based analysis is apparently strong.

One drawback of these tools is that they are unable to efficiently explore (i) very large models, (ii) models where the state transformation cannot be expressed in terms of simple modifications to state variables, or (iii) models where recognition of a death-state is complex. For example, model sizes become large any time one desires a detailed analysis of a detailed model; state transformations become complex if recovery transitions involve non-trivial computations, such as finding new routes for messages through a fault-tolerant network; death-state recognition may be complex if system operability is defined in terms of the system's ability to provide some service, e.g., every pair of operable processors are able to communicate using some specific routing protocol. We later give examples of all three situations.

This paper describes methods we have used to address these situations, and a software tool called ASSURE that embodies these methods. ASSURE combines the functions of ASSIST and SURE. The user's interface to ASSURE is an enhanced version of the ASSIST [2] language. ASSIST's power of expression is extended to almost arbitrarily complex models by allowing the user to write C language routines to recognize system failure, to recognize system transition conditions, and to express system state modification following a transition. Other techniques we describe are related to using the SURE bounds to efficiently analyze some large models. One method is to concurrently generate and analyze a model's state-space via depth-first-search (DFS) exploration. Memory requirements are limited to that needed to manage the DFS stack, instead of the entire state space (as is presently required with ASSIST/SURE); however, memory efficiency comes at the price of state recomputation. The method has the intentional and important advantage of supporting parallel processing on ordinary networks of workstations. We also investigate user-assisted methods for trimming the model space. Even with the fore-mentioned features, the sheer size of state-spaces involved in some models prohibit an exact and exhaustive analysis. To address this problem we have developed efficient ways of jointly using Monte Carlo simulation and the

SURE bounds to construct confidence intervals on estimated upper and lower reliability bounds. In addition, our method supports estimation of arbitrary measures of system performance in death-states, and we have extended the ASSIST language to support automated estimation of these user defined statistics. Finally, the Monte Carlo analysis is easily parallelized as well, again on a network of ordinary workstations.

ASSIST/SURE is only one of many good reliability tools; i.e., see the recent survey [8]. Various of the features we've incorporated into ASSURE have been used in the past by other tools. The notion of expressing models in a high level language and then automating the generation and analysis of the underlying Markov chain is common to all modern reliability tools. For example, HARP [5] uses a fault tree description of failure processes and a petri-net description of recovery processes. From these a Markov chain is constructed and analyzed to provide system state probabilities. SAVE [7] uses a language describing a machine-shop with repairmen. SHARPE [17] provides a number of different model types, in a sort of analysis toolbox. The notion of truncating a state-space (while developing it, or searching it) is found in the tools above, as well as in [6]. The idea of using a common programming language as a vehicle for describing a model is exploited in DEPEND[9], which also uses Monte Carlo simulation, as does SAVE [7]. A Monte Carlo version of HARP has also been developed [1]. Our intent is to show how ASSURE's features together allow us to attack very large and complicated system models, and to demonstrate a single tool that seamlessly allows either an exact analysis or a simulation analysis, and/or a serial solution or a parallel solution from a common (but general) model description. Our main contributions are implementation methods suitable for solving such models. These contributions are three-fold. First, we demonstrate that on an interesting set of large problems there is much to be gained by regenerating states in a depth-first analysis, rather than saving each generated state against the possibility that it will be visited again. This style of analysis permits solution of some models considered to be "out of reach" at the time [8] was written (i.e., 10^5 states, 10^{10} ratio of repair rates to component failure rates). One should note, however, that the relative advantage of the method decreases as the number of failures required to push the system into a death-state increases. Consequently, exact analysis using the method is best suited for systems that tolerate 2-5 failures in the mission time. We demonstrate empirically that this approach is ideal for parallel processing—a new and highly practical aspect of reliability analysis. Thirdly, we show how the SURE bounds lend themselves to an efficient Monte Carlo analysis, which itself is parallelizable.

It might be argued that detailed analysis of large models is unnecessary, since at some level a reliability model will have to mask details anyway, and an expert modeler can often craft a good model from a detailed understanding of the system being modeled. While we will never dispute the power of a expert modeler using a simple tool, we believe that the need to analyze large detailed models is inevitable. We anticipate the day when a system is specified and designed using a single tool from which reliability and performance analyses are automated. An automatically

generated model is far more likely to be large and complicated than one developed by a human expert. Furthermore, an automated analysis accommodates a system design or parameter change by simply redoing the analysis—that same change may invalidate a human expert's entire approach. Towards this end, we are exploring ways in which large complex models might be automatically analyzed.

It might also be argued the SURE approach is inadequate, owing to its assumption of time-independent failure rates. While this argument has some validity, we are not attempting to advance any particular side in the sometimes heated debate over reliability tools. We believe that the techniques we describe are not limited to SURE; they can be applied to any mathematical analysis based on paths through a state-space. Perhaps the potential shown by ASSURE for large problems may motivate mathematical research on path-based analysis that overcomes SURE's limitations.

This paper is organized as follows. Section 2 describes two model problems that exhibit challenging characteristics. Section 3 describes extensions we've provided for the ASSIST language to enhance model expression. Section 4 presents the SURE bounds. Section 5 describes implementation techniques that support the analysis of large complex models, and Section 6 explains how SURE bounds can be used in the context of an efficient Monte Carlo analysis. Section 7 presents our conclusions.

2 Two Examples

Our work has been motivated in large part by the challenges presented by two diverse yet representative reliability models. The first model is of a fault-tolerant flight-control computer network having a complex recovery mechanism, the second is that of a large computer network that achieves fault-tolerance through redundancy of communication channels. This section discusses both models, and the characteristics which challenge the capabilities of existing SURE-based tools.

The first problem presents the challenge of state-space size, and complexity of expressing a complex reconfiguration strategy within the confines of the modeling language. These challenges are both present in a model based roughly on AIPS [12], an architecture developed by Stark Draper Labs. The model is comprised of a Fault-Tolerant-Processor (FTP), that manages a collection of "devices" (sensors). The devices are replicated four times for quad redundancy, and are distributed across two networks, accessed by the FTP from six channels. Only selected links in the network are "in use" at any time. The set of selected links in a network establish a virtual bus between one FTP channel, and every operational node in the network. In the event a selected link or a network node fails, the network is considered to be down. However, it may be repaired if another set of links can be found to establish the virtual bus. During recovery the FTP knows to ignore the downed network, and to take its sensor data from the other network. The system is considered to have failed if the FTP itself fails, if both networks are simultaneously down, or if the majority of operable devices of any type are not able to communicate with the FTP. Figure 1 illustrates an

example of this network and its hardware components. Shown are four channels linking the FTP to the networks, six network interfaces, thirty-two links, fourteen switching nodes, eight interface devices, and sixteen devices. "In-use" links in one particular system state are highlighted; a number of links are shown to have failed.

The particular set of links chosen during repair to re-implement a virtual bus will impact the distribution of the remaining time until system failure, especially if failure rates of remaining components are heterogeneous. Greater accuracy is obtained then by explicitly modeling the reconfiguration process than by assigning an approximate recovery rate to a network failure. If we accept the desirability of an accurate recovery model, we consequently require that the reliability tool be able to concisely express the reconfiguration strategy.

The second problem arose in a study comparing the effectiveness of fault-tolerant routing protocols on a binary hypercube. Nodes and links may fail; when one does, no explicit recovery is attempted. However, network messages can accommodate such failures by adaptively rerouting around failed nodes or links. A variety of fault tolerant routing protocols exist, some of which may not find a extant route. Given a protocol, the system is considered to have entered a death-state if either more than half of the nodes have failed, or if there exist two operable nodes between which the protocol cannot establish a message path. The complexities of these protocols defeat more elegant graph-based based reliability analyses, and we are left to use simulation if we are to estimate reliability.

This model presents us with two fundamental problems. First, depending on the network size, tens to hundreds of link and node failures can be tolerated before the system enters a death-state. The size of the state space absolutely prohibits an exhaustive analysis. The second problem is that recognition of a death-state is expensive. Given the state of the network, one must essentially simulate message routing behavior between every pair of nodes. The cost of a single connectivity check is $O(L)$, implying an $O(LN^2)$ death-state recognition cost.

The sections to follow describe the methods we've used to address the challenges posed by these problems.

3 Language Extensions to ASSIST

The ASSIST language (see [2]) provides a simple means of describing a system and how it evolves in the presence of failures and recoveries. The notion of state variable is central to ASSIST; one of the first roles of an ASSIST model is to declare the state variables (and their initial values) just as variables are declared in programming languages. Evolution of the system is described in terms of Boolean conditionals on the state variables (describing conditions under which a transformation may occur), and simple modification of state variables (describing the transformation itself). For example, a state variable `N` may describe the number of working processors, any of which may fail. The ASSIST statement

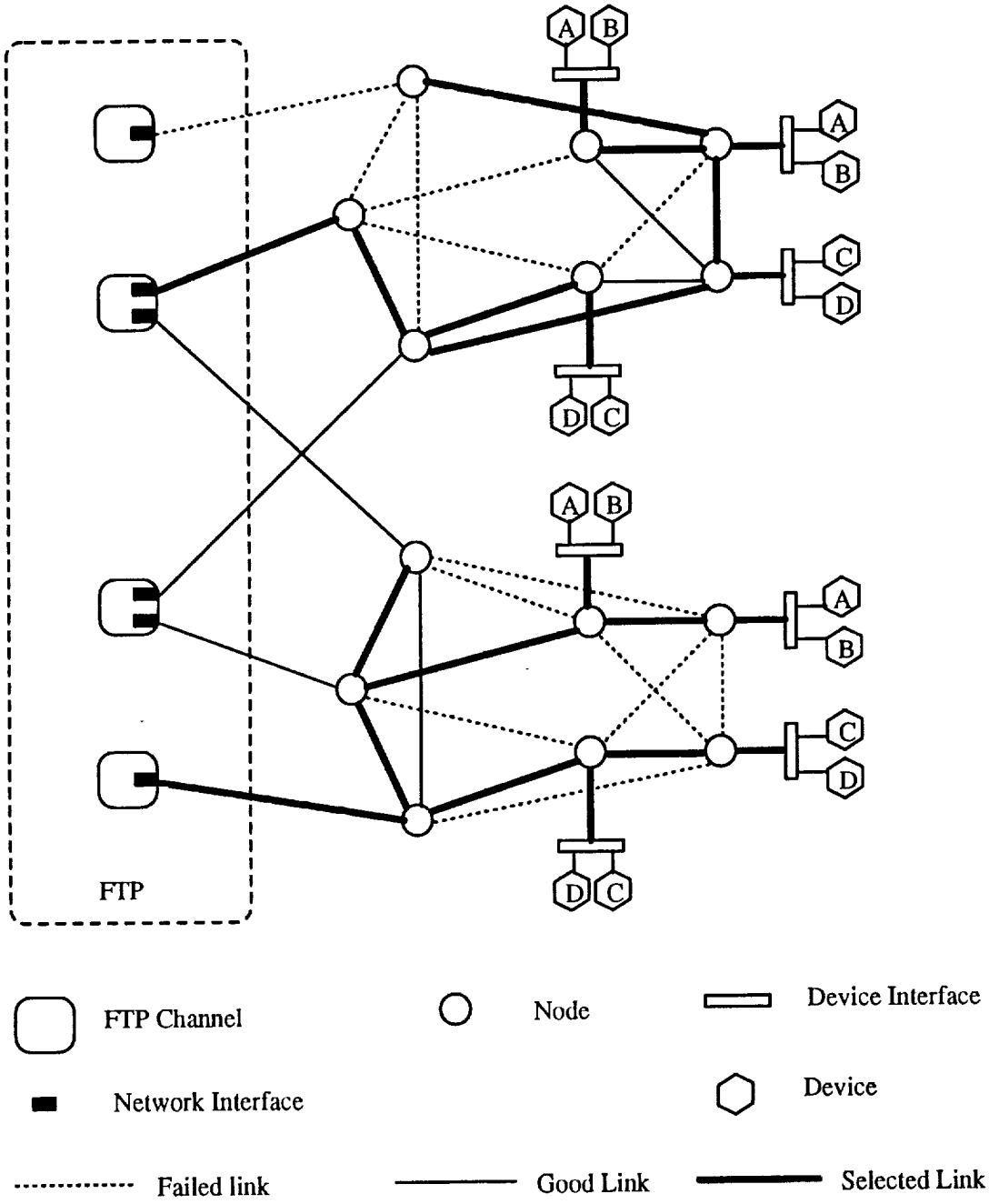


Figure 1: Example of reconfigurable flight control computer network, highlighting virtual bus connections.

```
IF N>0  TRANTO  N=N-1  BY N*LAMBDA;
```

declares that from any system state where N exceeds zero, another processor can fail, and change the system state by decrementing N. The mathematics of SURE assume that a component's lifetime

is exponentially distributed; the statement above declares that the transformation occurs with rate N*LAMBDA (LAMBDA is defined as a constant elsewhere). Boolean conditionals also identify death-states, for instance,

```
DEATHIF N=0;
```

declares that the system is in a death-state whenever all processors have failed.

This particular example is unrealistically simple. Larger ASSIST models employ compound Boolean expressions as conditionals, and modify several state variables as a result. For instance, the statements below were taken from a working ASSIST model.

```
(* COVERAGE *)
 DEATHIF (FT[1]+FT[2]+FT[3]+FT[4] ) <=
        (FF[1]+FF[2]+FF[3]+FF[4] );

(* EXHAUSTION *)
DEATHIF FT[1] + FT[2] + FT[3] + FT[4] < 2;
DEATHIF NI[1] + NI[2] + NI[3] + NI[4] + NI[5] + NI[6] < 1;

(*  TRANSITION RULES  *)
FOR I=1,6;
  IF NO1G[I]=1 TRANTO NO1G[I]=0, P[1]=0, CT=CT+1 BY LNO;
  IF NO2G[I]=1 TRANTO NO2G[I]=0, P[2]=0, CT=CT+1 BY LNO;
ENDFOR;
```

Here we see that ASSIST allows arrays of state variables, multiple DEATHIF and TRANTO statements, and looping constructs. An important aspect of ASSIST models is that they are essentially algorithmic. The TRANTO statements give a set of rules; any time the system state satisfies a rule, a transition from that state is possible. The statements following the keyword TRANTO describe how the system state is correspondingly modified, and the statement following keyword BY gives the transition rate.

We found that the ASSIST syntax for describing state modification was too limited to efficiently express the dynamic network reconfiguration required by our first model problem. There, given the operational status of network links, nodes, and devices, we must apply an algorithm to find a subset of these components that form a bus. Nevertheless, we saw that it was still possible to exploit the essential idea behind ASSIST, which is to express state transitions in terms of recognizing when and how they occur. Our simple extension is to allow the statement following a TRANTO to be a call to a subroutine in the C programming language, where declared ASSIST state variables may be both read and written directly. Similarly, the ability to express DEATHIF and TRANTO conditions is extended by allowing calls to C routines that analyze the variables of the present system state and

return a Boolean value indicating whether a particular condition is satisfied. To our knowledge, ASSURE is the only tool which both provides an analytic solution (as opposed to only simulation), and allows manipulation of model state variables by a general programming language. In our experience this ability proved invaluable when describing complex reconfiguration strategies, and when analyzing models with complex death-state conditions. In support of these extensions, we also allow a user to write a subroutine to compute the initial system state variable values, and to build static C data structures (which ought only to be read, not modified) for use by other routines. For example, we've used this feature to describe static network topologies and let the system state vector contain only the operational status of each component.

These extensions are conceptually simple, and are implemented by using an ASSURE-to-C source code translator. The translator parses the ASSIST model, translates references to ASSIST state variables into references to C variables, and uses the ASSIST model structure to create problem-dependent C subroutines for detecting death-states and for generating all transitions possible from a given state. These subroutines are compiled and linked to pre-compiled problem-independent code that controls the generation process and performs the SURE analysis. On most models, the translation step requires a few seconds and the compilation/linking step requires a few tens of seconds, on ordinary workstations. This relatively small front-end cost is easily amortized when a large model's execution phase takes minutes, or longer.

4 The SURE Theorem

Subsequent discussions are better understood following a brief description of the SURE theorem. A fuller treatment of these bounds are given in [3].

We may think of a semi-Markov state-space as a directed graph whose nodes represent states, and whose edges represent transitions. A precise mathematical definition can be found in many standard texts, e.g., [16]. The SURE theorem applies to semi-Markov processes with two types of transitions. *Slow* transitions are exponentially distributed, with small transition rates as compared with the *fast* transitions, that may have general distributions. Slow transitions typically model hardware component failure, whereas fast transitions model repair processes. The difference in transition rates may span several orders of magnitude.

The sequence of transitions defining a path through a semi-Markov state space reflect a possible system behavior in time. The amount of time the system takes to traverse a given path p is random, call it S_p. Given a *mission time* T, the SURE theorem gives formulae for upper and lower bounds ($U_p(T)$ and $L_p(T)$, respectively) on $\Pr\{S_p \leq T$, path p is taken$\}$. These bounds are of particular interest when the last state on p is a death-state.

Let $\mathcal{D}$ be the set of death-states, let I be the initial system state, and let $\mathcal{P}$ be the set of all paths from I through states not in $\mathcal{D}$, to some member of $\mathcal{D}$. The probability that the semi-Markov

process enters $\mathcal{D}$ within time T is

$$\Pr\{\text{Death state entered within time } T\} = \sum_{p \in \mathcal{P}} \Pr\{S_p \leq T, \text{ path } p \text{ is taken}\}. \tag{1}$$

To use the SURE bounds one discovers and analyzes every path in $\mathcal{P}$ (at least the ones with sufficient probability) as follows. We classify every state on a path p as being a class 1,2, or 3 state. A state is in class 1 if its transition on p is slow, and every other transition from the state is also slow. Any state whose transition on p is fast is in class 2; the transition from a class 3 state is slow, and there is at least one fast transition from that state. The following class-specific parameters are needed to state the SURE bounds.

Class 1 Let k be the total number of class 1 states on p. For the i^{th} class 1 state define λ_i to be the rate of the transition out of the state, and define γ_i to be the sum of rates of all other transitions from that state.

Class 2 Let m be the total number of class 2 states on p. For the i^{th} class 2 state define ϵ_i to be the sum of rates of all slow transitions from it. Let ρ_i be the probability that the particular transition on p is successful (as opposed to some other transition from that state); let $\mu_{2,i}$ and ρ_i respectively be the conditional mean and standard deviation of the state holding time, given that the selected transition on p is successful.

Class 3 Let n be the total number of class 3 states on p. Let α_i be the rate of the transition out of the i^{th} class 3 state on p, and β_i be the sum of rates of all other slow transitions from that same state. Define $\mu_{3,i}$ and $\sigma_{3,i}$ to be the mean and standard deviation of the holding time in that state, given that a fast transition occurs (instead of the slow transition that did occur).

Finally, let $Q(T)$ be the probability of traversing by T a path constructed by concatenating the k class 1 states, and let $r_1, r_2, \ldots, r_m$, and $s_1, s_2, \ldots, s_n$ be strictly positive numbers such that $T > \Delta = r_1 + r_2 + \cdots + r_m + s_1 + s_2 + \cdots + s_n$. Then

$$L_p(T) \leq \Pr\{S_p \leq T, \text{ path } p \text{ is taken}\} \leq U_p(T)$$

where

$$U_p(T) = Q(T) \prod_{i=1}^{m} \rho_i \prod_{j=1}^{n} \alpha_j \mu_{3,j} \tag{2}$$

and

$$\begin{aligned} L_p(T) &= Q(T-\Delta) \prod_{i=1}^{m} \rho_i \left[1 - \epsilon_i \mu_{2,i} - \frac{\mu_{2,i}^2 + \sigma_{2,i}^2 + \rho_i^2}{r_i^2} \right] \\ &\quad \times \prod_{j=1}^{n} \alpha_j \left[\mu_{3,j} - \frac{(\alpha_{3,j} + \beta_{3,j})(\mu_{3,j}^2 + \sigma_{3,j}^2)}{2} - \frac{\mu_{3,j}^2 + \sigma_{3,j}^2}{s_j} \right] \end{aligned} \tag{3}$$

Computation of the α, μ, σ, and ρ values is standard. The following suggestions for r_i, s_i, and bounds on $Q(T)$ are given in [3]:

$$r_i = \left(2T(\mu_{2,i}^2 + \sigma_{2,i}^2)\right)^{1/3}$$

$$s_j = \left(\frac{T(\mu_{3,j}^2 + \sigma_{3,j}^2)}{\mu_{3,j}}\right)^{1/2}$$

$$\frac{\prod_{i=1}^{k}(\lambda_i T)}{k!}\left(1 - \frac{T}{k+1}\sum_{i=1}^{k}(\lambda_i + \gamma_i)\right) \leq Q(T) \leq \frac{\prod_{i=1}^{k}(\lambda_i T)}{k!}. \quad (4)$$

An important characteristic of these bounds is that they depend only on a small amount of information pertaining to the path. In fact, the products in Equations (2)–(4) can be accumulated in a small, fixed amount of storage space as a path is extended. For computational reasons (for $Q(T)$) we do separately save the λ_i and γ_i values from each class 1 transition, but this requires the storage of only two floating point numbers per transition.

One way to use these bounds is to explore all paths from I to $\mathcal{D}$. Whenever a path $p \in \mathcal{P}$ is discovered, $L_p(T)$ and $U_p(T)$ are computed and added to accumulating totals $L(T)$ and $U(T)$. It is important to prune loops, or other paths with very small (relative) probabilities. SURE-based tools typically prune a path p once $U_p(T)$ is smaller than some threshold ϕ (which may be given by the user, or can be found automatically). Upon pruning p, $U_p(T)$ is added to an accumulating total $P(T)$; the final lower and upper bounds on system failure by time T are then $L(T)$ and $U(T) + P(T)$. One typically desires to find ϕ such that $P(T)$ is an order of magnitude smaller than $U(T)$.

A user of the original ASSIST/SURE toolset constructs a state-space using ASSIST, and analyzes it using SURE. In the next section we describe how the generation and analysis can be combined, and how the whole process is easily parallelized.

5 Analysis Techniques

This section describes ASSURE's technique of depth-first generation and analysis of a model, parallelization of this method, and a user-assisted technique for trimming the model during its generation and analysis. We demonstrate empirically that these techniques effectively accelerate the solution time of some large ASSURE models.

5.1 Depth-First Generation and Analysis

Memory usage seriously degrades the execution time of ASSIST and SURE on very large state-spaces. Not only may tens of megabytes be required to store the model, but both the generation and analysis processes may suffer thrashing in a virtual memory system.

We can address the problem by trading off computational efficiency for space efficiency. ASSIST stores all generated states; upon creating a state it looks to see if that state already exists, and extends a path through that state only upon its initial discovery. A different approach is to simultaneously generate and analyze the state-space along a path, and to discard discovered states once they are no longer needed *for that path.* This provides a significant memory savings since memory requirements are proportional only to path length times fanout. The price paid for memory efficiency is the recomputation of state descriptions. This tradeoff works to our advantage for an important class of problems. As we will see, on the large examples we have studied the benefits of memory efficiency are evident. Furthermore, the approach lends itself to parallel processing (which was our initial consideration) because distinct paths can be generated and analyzed separately on different processors. However, the approach has its limitations. Best results are obtained when the system of interest tolerates only a few number of failures within the mission time, say, 5 or fewer. Beyond that, the combinatorics of the approach threatens to create unacceptable solution times.

Our tool ASSURE combines the functions of ASSIST and SURE as follows. A path p is represented internally by a data structure we call a *path-record.* A path-record contains a copy of every ASSIST state variable, whose values represent the last state on the path. A path-record also contains a list of the λ_i and γ_i values of all class 1 transitions on the path, and accumulated products for Equations (2)–(4). ASSURE begins by initializing a path-record to reflect I, and places it on a *working list.* ASSURE enters a loop where the first path-record on the working list is removed and $U_p(T)$ is computed and compared against the pruning threshold. If $U_p(T)$ is sufficiently high, the path-record's state variables are checked against all death-state conditions. The code that performs this check is C code translated from ASSIST DEATHIF statements. A path-record that survives pruning and death-state testing is subjected to extension through all possible transitions, by checking its state variables against every TRANTO condition specified in the ASSIST model. Every time a TRANTO condition evaluates to **true** a copy of the path-record is created, its state variables are modified as proscribed by the ASSIST model, and the value of the transition rate specified following the transition's BY keyword is recorded. Again, these tests and modifications are performed by C code translations of ASSIST model statements. By testing the path-record against all TRANTO conditions we discover and generate all transitions possible from the path-record's last state. Given these transitions and their rates, all the quantities needed by the SURE bounds for each new path are computed, and recorded in each new path-record. The new path-records are attached to the head of the working list, and the process continues until the working list is empty.

The description above shows that ASSURE generates all sufficiently probable paths from I to $\mathcal{D}$ via a depth-first generation and analysis strategy. In addition, ASSURE provides the additional capability of determining whether a model can survive any K failures without entering a death-state. This is easily encorporated by recording the number of slow transitions on the path, and

prune once that count reaches K. If no death-states are uncovered, then the system model survives any combination of K failures.

It is important to observe that the techniques described above do not depend on the specifics of the ASSIST language. Any formal description of a reliability model will do, provided that one can automatically and quickly find all transitions and their rates from any given state system state. Indeed, as a follow-on to ASSURE, we have built an object-oriented language and tool, REST, that is based on these same principles [15]. Within that framework we have also written a SAVE-to-REST translator, thereby providing transient SAVE models with the computational advantages described in this paper. Furthermore, we believe other tools could also incorporate such an approach. For instance, HARP is widely used, but encounters memory problems on large models [18]. Since HARP analysis is based on a Markov chain, and since system death conditions are recognizable from the defining fault-tree, one could apply a depth-first combined state-space generation and analysis method as we have done with ASSURE. Upon reaching a death-state or a pruned state one could examine the path and numerically compute the exact probability (by uniformization [16]) of reaching that state by time T.

At any time, the memory requirements of ASSURE are basically those of storing the working list. However, ASSURE ends up doing more computation to generate the state-space than does ASSIST. The tradeoff often works to ASSURE's advantage. On moderately large ASSIST models of our first model problem (models that lack complex reconfiguration), ASSURE runs ten to twenty times faster than does ASSIST/SURE. A simple analysis helps to quantify the tradeoff. Component failures essentially drive changes in a system state. Let X be a state-vector with N components, and suppose that any given collection of failures results in the same state of X regardless of the sequence in which the failures occur. Ignoring effects of possible aggregation (i.e., different collections of failures resulting in the same state), a state s defined by j failures will lie on $j!$ different paths. But s has j immediate predecessors, implying that ASSIST will discover s exactly j times. To a first approximation then, if the model tends to tolerate j failures before entering a death-state or being pruned, ASSURE does $j!/j = (j-1)!$ times more computation than does ASSIST. On the other hand, ASSURE's memory requirements are small enough that it tends to operate without page faults, whereas ASSIST is observed to thrash on large models. We estimate that ASSIST's average cost of "touching" a state is several hundred times higher than ASSURE's. These estimates suggest that ASSURE is more efficient than ASSIST when the system model tolerates a handful of errors within the mission time, say, 6 or fewer.

5.2 Parallelization

ASSURE's generation and analysis technique is highly suitable for parallel processing, because processors can independently generate and analyze distinct paths from I to $\mathcal{D}$. One way is have a controller process generate all one-step paths $p_1, p_2, \ldots, p_N$ from the initial state I, then distribute

these path-records among processors to seed their working lists. Once seeded, a processor is free to execute exactly as in the serial case. Each processor then independently accumulates a pruning bound, and SURE bounds. The overall bounds are obtained by adding the contribution from each processor.

The method above has the disadvantage that one processor may complete its work long before another processor does. We have used two different methods of dealing with this problem. ASSURE has been parallelized on a 32-processor Intel iPSC/860 multiprocessor. This machine has a fast communication network, making it feasible for a processor with excess path-records to send some to deficient processors. We implemented and studied several dynamic load-balancing schemes that balance the number of path-records per processor nearly perfectly when called. Our studies found that for ASSURE problems it didn't matter very much *how* the load was rebalanced, so long as it was rebalanced. A discussion of the different schemes studied is given in [14].

This type of balancing is not suitable for a loosely coupled network of workstations. However, load-balancing here is simple if the workstations share a common file system. The trick is to have every workstation generate path-records for every descendent of the initial state, and enumerate them the same way. Every workstation i begins by seeding its working list with descendent i; some dedicated process writes the number of unassigned descendents into a commonly viewed file. A workstation executes independently until its working list is empty, at which point it consults the remaining descendent count file to look for more work. If the value in the file is non-zero, the workstation decrements the count, and reseeds its working list with the appropriate descendent. Otherwise, the workstation writes its own results into a reserved file. The computation is complete once all workstations have attempted and failed to acquire additional workload. A monitoring process accumulates and reports the individual workstation results. ASSURE does all this automatically, given a run-time option specifying the network names of machines to use.

The simplicity and power of parallelizing SURE bounds calculations gives us reason to believe that extremely large models can be generated and analyzed in a reasonable amount of time. For example, we considered two different ASSURE models of our example reconfigurable network problem. The first, called NetA, has 61 elements in its state-vector. Recovery from network failure is simplified enough to be expressed in pure ASSIST. The second model uses our language extensions, and has 83 elements in its state-vector. Each hardware component (channel, network interface, link, node, device interface, device) has its own failure rate; the ratio of the fastest recovery rate to the slowest failure rate is 10^{10}. In the data reported below, a path-record was pruned as soon as the upper bound on its probability dropped below $\phi = 1e\text{-}15$. The analysis of NetA generated 3.6 million nodes with an average number of failures when a path terminated of 3.9. If we estimate the actual size of the state-space explored as W choose L (i.e., $W!/((W-L)!L!)$) where W is the length of the state vector and L is the number of component failures, then the NetA analysis is of approximately 0.5 million unique states. NetB generates 57.5 million nodes, with 4.0 average failures

Model	1 Sparc	6 Sparcs	12 Sparcs	18 Sparcs	1 i860	32 i860s
NetA (3.6M nodes)	22 min	4 min	2 min	2 min	2.66 min	0.15 min
NetB (57.5M nodes)	7.5 hr	75 min	35 min	24 min	79.5 min	2.5 min

Table 1: Timings of first model problem on parallel platforms

at termination, and an estimated true state-space size of 2 million states. We report experiments conducted on 32 processors of the Intel iPSC/860 multiprocessor (based on the i860 CPU), and on a local area network using 6, 12, and 18 SUN Sparc workstations of various models. The iPSC/860 was dedicated to the application, whereas the network runs were competing with everything else on the network (which was lightly loaded) at the time. Also, our network timings have a resolution of only one minute, being taken from last-modification times on files. Table 1 presents these results.

The primary conclusion we draw from these timings is that the parallelization techniques work to dramatically reduce solution time. Furthermore, while the performance shown is nearly an order of magnitude faster on a dedicated multiprocessor, one can still get impressive performance from workstation networks commonly found in research labs.

5.3 Model Trimming

The combinatorial growth of models explored by our method encourages us to search for ways of reducing the number of states generated and analyzed. For instance, consider a path that has undergone j failures, and suppose we could bound the probability of entering a death-state following any two additional failures. Instead of generating the model for an additional two levels we might trim it, and accumulate the death-state probability bound in the pruning sum. This mechanism could reduce the model size by a factor of as much as $(j+1)(j+2)$. We have developed a way of doing exactly that, and observe significant reductions in the model size. The method is a generalization of the notion of a *trimming bound*, described in [19].

Consider all slow transitions out of an arbitrary state X. Typically each one is related to the failure of a component or to a set of components. Some transitions may lead immediately to death-states; call these transitions *unsafe*. At the other extreme, there are transitions which may also be taken from the next state if not taken from the present one, and which are guaranteed to be "not-unsafe" from the next state; call these *safe*. For example, safe transitions are defined whenever one has a collection of components and spares that can tolerate a component failure by using a hot spare to immediately replace it. Such a transition is safe because it may occur in any state and not cause system to fail. Finally, we call a transition *conditionally safe* if the system does not enter a death state by taking it, nor will a death-state be entered if a safe transition is taken first.

Our method is different from [19] in that we make a distinction between safe and conditionally

safe transitions. Like the earlier work, we assume (i) that components fail at a low constant rate, (ii) fault recovery depends only on the time since fault occurrence, and (iii) all transitions to system failure are component failure transitions.

Now from any state X, let U(X), C(X) and S(X) be the sum of rates of unsafe, conditionally safe, and safe transitions, respectively. Also, let R(X) be the sum of rates of exponentially distributed recovery transitions from X, and let M(X) be an upper bound on the sum of slow transition rates in any state reachable from X in two transitions. To construct a trimming bound we may consider the behavior of a simple Markov chain shown in Figure 2. From X it describes an aggregate recovery, an aggregate unsafe transition, aggregate conditionally safe transition, and aggregate safe transition. The chain also expresses a second level of behavior, with unsafe and not-unsafe transitions. The rates on the second level transitions are upper bounds on the aggregate rates in the actual system. The effect of recovery transitions on these states are omitted, which serves to accelerate the simple chain towards failure state F even faster than the actual system. Our trimming bound is given by adding the SURE upper bounds on each of five paths which extend the path to X further to the failed state F. This sum is greater than the sum of probabilities of reaching any death state eventually reachable by taking a failure transition from X.

In theory one could use the trimming bound by comparing it to a threshold ϕ (like the pruning threshold). If ϕ is larger, the only transitions from X that are generated are the recoveries, and the trimming bound is added to the accumulating pruning bound. In practice it is difficult for a general tool such as ASSURE to automatically compute the necessary failure rates (note, however, that this is less of a problem with tools that impose more structure on their model input description, from which the rates might be inferred). We've addressed the problem in ASSURE by allowing a user to write a C language function that computes $U(X)$, $C(X)$, $S(X)$, $R(X)$, and $M(X)$ for any state X. ASSURE then automatically invokes and uses the results of the routine. This mechanism allows a modeler to exploit knowledge of the system structure in order to quickly compute these transition rates, or upper bounds upon them (or a lower bound on $R(X)$).

Consider our first example problem. The ASSURE model defines the system to fail if any of the following conditions holds.

- A fault occurs in one network partition while the other partition is under repair.
- The number of FTP channels that are "good" (operable and in use) is zero, or is equal to the number of channels that are operable but are involved in a network repair.
- For every device type, the number of devices that are "good" is zero, or is equal to the number of channels that are operable but are involved in a network repair.
- A failed network is unable to establish a virtual bus to operative devices.

Using this information, one can write a routine that examines the model state variables and classifies the effect of every component failure as being safe, conditionally safe, or unsafe. Since the

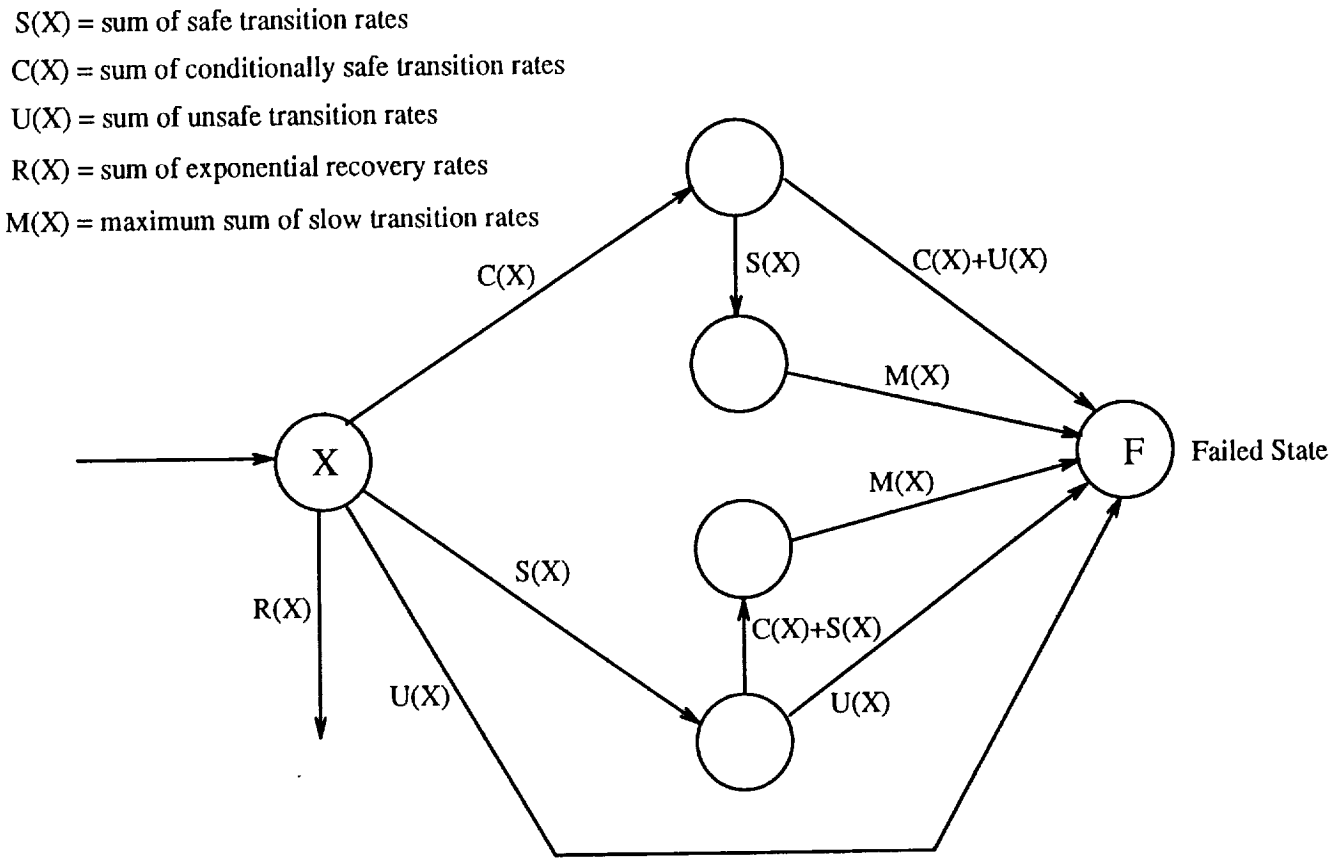


Figure 2: Markov chain used to construct the trimming bound

classification will be done often it is important to do it quickly. One can always misassign a transition to a class with less safety, e.g., assign what is actually a safe transition to the conditionally safe class. The bounds needed by ASSURE are obtained by summing rates within a class. For example, suppose X reflects a state in our model NetB where one partition is under repair. Then every transition related to a component failure that might trigger a network recovery in the *other* partition is classified as unsafe, e.g., the failure of a link on the virtual bus. On the other hand, transitions related to FTP channel failures may be in any of the three classes. If there is only one good FTP its failure will cause system failure, and hence that transition is unsafe. If there are two good FTP channels, and two failed channels then each FTP channel failure transition is conditionally safe; with three good FTP channels and one failed channel each transition is safe. Other component failures may be similarly analyzed.

A simple modification of this scheme deserves special comment. While ASSURE needs user assistance to produce U(X), C(X), S(X), and R(X), it does not need help computing M(X), provided that $M(X_i)$ does not increase along any set of states in any path $X_1, X_2, \ldots$. For this reason ASSURE provides an automatic trimming bound from state X where it is assumed that all slow

transitions from X are unsafe. When initiating any solution run, ASSURE can be told not to assume monotonicity.

To investigate the utility of these trimming options we solved the first model problem (NetB) using three different options, all with a trimming (or pruning) threshold of $\phi = 1e-15$. The first option we call *standard*—a path is pruned if the upper bound for that path is less than ϕ. We call the second option *monotonic*—this method uses the automatic monotonic trimming method, and needs no user assistance. The third option we call *user-assisted*, because the user supplies a routine that computes and classifies transition rates. The table below illustrates the results, noting the total number of states generated, the pruning bound, and the time required for solution on a single Sparc 1+ workstation. All methods obtained the same unreliability bounds for a 3 hour mission, 3.77e-9 and 3.96e-9.

Method	Total Number of States	Pruning Bound	Execution Time
Standard	57.5M	9.3e-11	590 min
Monotonic	7.2M	4.3e-11	112 min.
User Assisted	1.1M	4.5e-11	17 min.

From this data we see the tremendous advantage of exploiting a monotonic property over not exploiting it, and the further advantage of providing user assistance. It is also interesting to note that the standard method's node execution *rate* is nearly twice as fast as the others, since it suffers no overhead to compute lookahead trimming bounds. However, the overhead of computing more advanced trimming bounds is clearly worth the effort.

The key ingredient to making the user-assisted bounds work well is that the user-supplied routine be able to quickly compute upper bounds on the transition rates. The alternative is to let ASSURE discover these rates (at least U(X) and C(X)+S(X)) by generating the descendents of X. We tested the alternative, and found no performance gains.

6 Simulation

Despite the promise of analyzing large state-spaces via parallel processing and smart trimming, the problem remains that gargantuan state-spaces defeat any approach based on exhaustive analysis. This is especially true in systems which tolerant many failures. Even if a tool can analyze a model, albeit slowly, a modeler may desire loose upper bounds on reliability in the course of exploring a model design. Alternatively, one may first wish to exhaustively test to ensure that any combination of K failures will not cause system failure, and then get a rough estimate of reliability. In such cases a Monte Carlo simulation approach can help. This section outlines such an approach, based on importance sampling. We first discuss the mathematics of sampling and show that the basic method is sound. We also point out that importance sampling based on SURE bounds achieves variance reduction over another standard method. We then consider parallelization, and observe excellent

speedups. Next we discuss optimized death-state checking, and also further language extensions to support general statistical measurements. Finally we discuss some important implementation considerations.

6.1 Mathematical Basis

For any path p ending in $\mathcal{D}$ (i.e., $p \in \mathcal{P}$), let $f(p)$ denote the probability that the system chooses p on its way to $\mathcal{D}$, if left to run sufficiently long. Then

$$\Pr\{\text{System failure by time } T | \text{path } p \text{ is taken } \} = \Pr\{S_p \leq T | \text{path } p \text{ is taken}\}.$$

Thus

$$\begin{aligned} \Pr\{\text{System failure by time } T\} &= \sum_{p \in \mathcal{P}} f(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \\ &= E_f[\Pr\{S_P \leq T | \text{path } P \text{ is taken}\}] \end{aligned} \tag{5}$$

where P is the random path chosen to $\mathcal{D}$. A Monte Carlo approach is to estimate this expectation via random sampling of $\Pr\{S_P \leq T | P \text{ is taken}\}$.

Given a path p and SURE bounds $L_p(T)$ and $U_p(T)$, we know that

$$\frac{L_p(T)}{f(p)} \leq \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \leq \frac{U_p(T)}{f(p)}. \tag{6}$$

This inequality could be used to estimate bounds on $E_f[\Pr\{S_P \leq T | \text{path } P \text{ is taken}\}]$, but there is a serious problem with such an approach. When P is sampled from f, from any state with both fast and slow transitions we will almost always chose the fast (recovery) transition. The majority of death-states occur in those rare cases when recovery mechanisms are defeated by low probability additional failures. Sampling paths using f means missing some of the death-states one is attempting to find. This problem has been recognized before [13, 4, 10, 8], where the notion of *importance sampling* is used. Intuitively, importance sampling is used to skew the path sampling towards rare events. Mathematically, let $g(p)$ be a different probability mass function for sampling paths such that $g(p) \neq 0$ whenever $f(p) \neq 0$. Then

$$\begin{aligned} E_f[\Pr\{S_P \leq T | P \text{ is taken}\}] &= \sum_{p \in \mathcal{P}} f(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \\ &= \sum_{p \in \mathcal{P}} \frac{f(p)}{g(p)} g(p) \Pr\{S_p \leq T | \text{path } p \text{ is taken}\} \\ &= E_g[R(P) \Pr\{S_P \leq T | \text{path } P \text{ is taken}\}] \end{aligned}$$

where $R(p) = f(p)/g(p)$.

To use importance sampling is to estimate the latter expectation by randomly sampling (with respect to g) bounds on $R(p)\Pr\{S_p \leq T | p \text{ is taken}\}$. From inequality (6) we see that for any path p

$$R(p)\frac{L_p(T)}{f(p)} \leq R(p)\Pr\{S_p \leq T|\text{path } p \text{ is taken}\} \leq R(p)\frac{U_p(T)}{f(p)},$$

or equivalently,

$$\frac{L_p(T)}{g(p)} \leq R(p)\Pr\{S_p \leq T|\text{path } p \text{ is taken}\} \leq \frac{U_p(T)}{g(p)}. \tag{7}$$

The Monte Carlo analysis consists of sampling (with respect to g) many independent replications of paths to $\mathcal{D}$, and for each computing $L_p(T)/g(p)$ and $U_p(T)/g(p)$ as samples. Following many replications we compute confidence intervals on $E_g[R(P)L_P(T)]$ and $E_g[R(P)U_P(T)]$, and use these to construct confidence intervals on the probability of system failure by T.

Many different ideas have been suggested for important sampling, e.g., see [10]. We have been successful with a strategy that partitions transitions from a state into slow and fast classes, chooses the slow class with some probability q and chooses the fast class with complimentary probability. Within a class a transition is chosen with probability proportional to its transition rate. For a given path p, $g(p)$ is computed as the product of the probabilities of each forced transition decision. This transition selection strategy was proposed in [13]. However, part of that proposal is to also sample holding times, conditioning them on no transition time exceeding T. When $\mathcal{D}$ is reached, the sample statistic is of the form $d(p)f(p)/g(p)$, where $d(p)$ is the product of ratios of the form $h(t_i|t_{i-1},k)/\hat{h}(t_i|t_{i-1},k)$. Here t_i is the sampled transition time from the i^{th} state on p, say k, given that k is entered at time t_{i-1}. $h(t_i|t_{i-1})$ is the density of that transition time using k's true holding time distribution, and $\hat{h}$ is the forced density function. Contrast the measure $d(p)f(p)/g(p)$ with the SURE-based measure $U_p(T)/g(p)$. A key point is that conditioned on taking path p, $d(p)f(p)/g(p)$ is still a random variable ($d(p)$ varies), whereas $U_p(T)/g(p)$ is deterministic. This immediately implies that the expected average measure in the original scheme has a larger variance does the expected average SURE measure. Therefore confidence intervals based on SURE bounds (when using the same transition selection strategy) will on average be smaller.

The quality of results obtained from importance sampling schemes are known to be sensitive to the problem class. We were naturally concerned whether the schemes we examined were effective on problems for which SURE was intended. Happily, the scheme above with $q = 0.5$ has proven to give results consistent with SURE analysis (this setting was also recommended in [4]). We tested the simulation-based results with SURE predictions, on a suite of problems used at NASA to validate ASSIST. Three of these are listed below, as well as models NetA and NetB, described earlier, using standard pruning. All simulation runs are based on 10,000 replications. The simulation-based lower and upper bounds are given as 95% confidence intervals, and timings are taken on a SUN Sparc workstation. This data suggests that the simulation based approach is able to find small intervals around the exact bounds, and in the case of the very large models do so more rapidly than the

Model	Size (nodes)	Exact Bounds	Solution time	Simulation Bounds (10,000 replications)	Solution time
aclust3	34623	4.94e-8, 4.95e-8	9 sec	(5.00 ± 0.14)e-8, (5.01 ± 0.14)e-8	31 sec
arcsxod1	1032	6.81e-3, 6.92e-3	0.4 sec	(6.63 ± 0.25)e-3, (6.74 ± 0.26)e-3	21 sec
billee	991	2.24e-7, 2.25e-7	0.4 sec	(2.25 ± 0.11)e-7, (2.25 ± 0.11)e-7	27 sec
NetA	3.6M	4.02e-6, 4.12e-6	22 min	(3.47 ± 0.51)e-6, (3.82 ± 0.57)e-6	2.2 min
NetB	57.5M	3.76e-9, 3.97e-9	7.5 hr	(3.68 ± 0.28)e-9, (4.04 ± 0.35)e-9	5.4 min

Table 2: Comparison of SURE-based and simulation-based analysis

exact analysis. However, it is also clear that orders of magnitude more replications are needed if we wished to shrink the confidence intervals to less than one percent of the mean. The advantage of simulation is that reasonably good numbers can be gotten relatively quickly. We expect there is utility in numbers known to be uncertain within 10%.

We also estimated reliability on the models above using skewed holding times as described in [13, 8]. On the small models there was no appreciable difference between the relative errors (confidence interval width divided by sample mean) of the two approaches. However, on NetA and NetB the SURE-based approached yielded relative errors that are 20% smaller. The SURE approach also runs 10–20% faster, since it avoids random number generation for holding times. For the 10,000 replications examined here, the confidence intervals for both approaches are not small enough to distinguish between an estimate based on SURE's upper bound, or the estimate of the precise probability.

The primary motivation for importance sampling is variance reduction. It is therefore instructive to examine how the sample variance achieved under our scheme changes as the class probability threshold q changes. This is illustrated in Figure 3, where for the NetB model we plot 95% confidence intervals on the upper bound 3.97e-9, following 10,000 replications. This data shows the danger of skewing q too far one way or the other, $q = 0.5$ appears to be a satisfactory setting. However, since effective importance sampling is known to be problem class dependent, ASSURE can call a user written routine to do the importance sampling. Such a routine is passed a description of the system state, and all transitions possible from that state (and their rates). The routine chooses a transition, and reports back the probability of making that choice under the importance sampling strategy. This is all the information ASSURE needs to correctly compute its statistics.

6.2 Parallelization

Simulation replications are trivially parallelized; we have done so on the workstation network. The only challenge is to use a load-balancing scheme that does not incur excessive overhead, but which

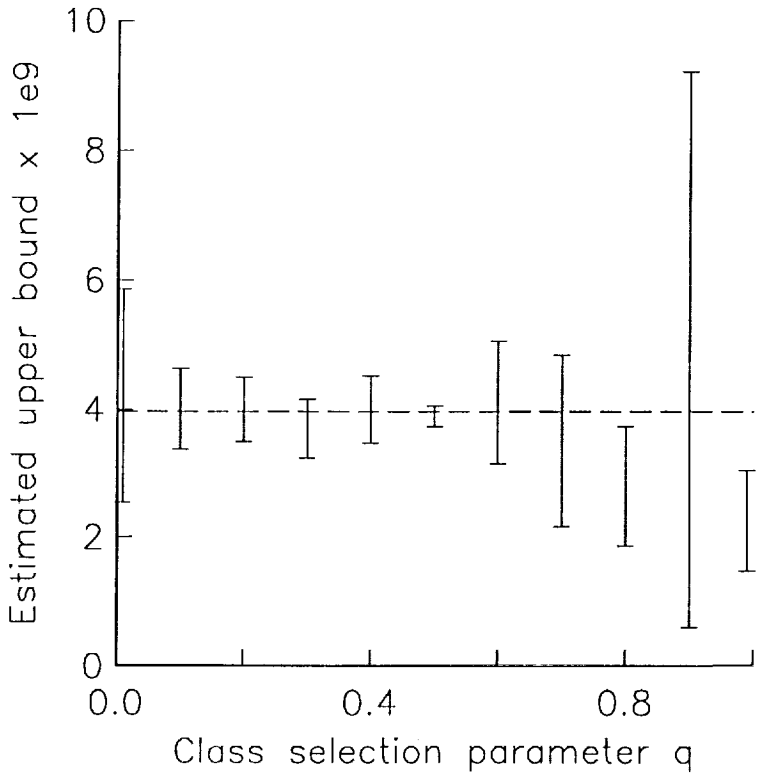


Figure 3: Confidence intervals as function of importance sampling parameter q

is responsive to changing network loads. Our scheme is to maintain a commonly accessed file, to contain the remaining number of replications. A workstation devoid of work accesses this file, acquires some fixed number G replications (G is user-defined so that replications are acquired no more than, say once a minute), modifies the file and releases it. The simulation is complete after every workstation finishes its work, and sees zero remaining replications. A monitoring process combines and reports the aggregate results. To demonstrate the effectiveness of this approach, we simulated NetB for 100,000 replications on 1, 6, 12, and 18 workstations at a time when the network load was low. The running times were respectively 35, 6, 3, 2 minutes. Once again we see the tremendous advantage offered by parallel processing.

6.3 Additional Issues

We now consider some auxiliary issues. In ASSURE, simulation-based analysis generates different problem-dependent code than does exact analysis; the generation of a state's descendents is done in two passes. The first pass identifies the *existence* of each descendent, and its transition rate. This pass does not actually perform the state-space modification. ASSURE's simulation control code selects a descendent, and in a second pass that descendent's state is created. We judged this approach to be crucial for problems with large state vectors, and/or complex state-modification

routines. Indeed, this approach yielded a factor of two reduction in execution time on NetA and NetB.

Our study of the second model problem led us to consider another implementation issue, that of death-state checking. ASSURE's basic scheme checks death-state conditions after every transition. This makes sense for many models, including all of the ones we've considered so far in this paper. However, consider our second model problem, where a system is considered to have entered $\mathcal{D}$ if there exist two operable processors that cannot communicate under the constraints of the fault-tolerant routing protocol. We noted earlier the high computational cost of checking that condition. The problem is that a path may be extended many times before reaching $\mathcal{D}$; most of the death-state checks are unnecessary, as they do not observe a death-state.

We exploit the fact that once the system state enters $\mathcal{D}$ it will not depart. The optimization is to only periodically check whether a path under expansion has entered $\mathcal{D}$, say, check every d transitions. We keep an ordered list of all path-records generated in the last d transitions. Upon reaching the d^{th} transition since the last check, we check the DEATHIF conditions on the present state variable values. If the state is not in $\mathcal{D}$ we release the first $d-1$ of the stored path records, and continue for another d transitions. Once a death-state is uncovered we must find the *first* state to enter $\mathcal{D}$ among the last d visited. Since their path-records have been saved in order of generation, we may perform a binary search. The number of death-state checks is thus approximately logarithmic in the path length, rather than linear. Observe that when systems can be repaired it may be possible to express a model that can pass into and out of $\mathcal{D}$, even though that may not be intended. For this reason, the optimization under discussion must be requested by a user, it is not automatic. However, one could adapt the scheme by always checking for a death-state on recognition of a recovery transition—we check the state just prior to the recovery and use that state as the terminus of a search interval.

In order to both illustrate the advantage of periodic checking and illustrate that simulation based analysis can handle large problems, we consider the second model problem. The routing protocol studied permits at most two "miss-steps", which means that the number of links crossed when i and j communicate is no larger then four plus the Hamming distance between i and j. With some straightforward tricks it costs $O(nodes \times links)$ time to determine whether a given network configuration is dead. We check the death-state condition every 100 transitions. Table 3 shows the effect on the simulation rate (replications/minute) of the "constant check" and "periodic check" methods, as the size of the problem increases. The table shows the problem size (in numbers of components), the average number of failed components when $\mathcal{D}$ is entered, and the simulation rates. This data clearly shows the advantage of periodic checking on problems of this type, and also shows that simulation-based ASSURE analysis is able to deal with relatively large problems, especially if we use parallel processing. On the largest problem shown here, we could expect to complete a 1000 replication run in approximately an hour using 18 workstations in parallel.

Hypercube Dimension	Size (nodes,links)	Avg. Failures	Constant Check sim. rate	Periodic Check sim. rate
6	(64,192)	73	4.52 reps/min	28.8 reps/min
7	(128,448)	166	0.4 reps/min	5.2 reps/min
8	(256,1024)	400	0.04 reps/min	0.92 reps/min

Table 3: Simulation rates on fault-tolerant routing problem, as the problem size varies

On this data the relative error from the SURE-based approach is approximately 33% smaller than that using skewed sample times, showing again the variance-reduction advantage of using SURE bounds.

Needs of the second model problem also gave rise to another language extension. The users were interested in obtaining statistical information about the system configuration in death-states, e.g., the average number of failed nodes and/or links. It was relatively easy to provide this by allowing "statistics" variables to be declared in the ASSIST model, e.g.,

```
SAMPLE FailedNodes;
```

A user provides a routine, called when a death-state is recognized, that assigns values to all `SAMPLE` variables. ASSURE automatically computes averages and confidence intervals, reporting these at the end of the analysis.

7 Conclusions

This paper demonstrates methods for accelerating the solution time of reliability analyses based on the SURE bounds. Our methods are centered around the notion of simultaneously generating and analyzing a state-space along a failure path, but discarding the state information once the path is analyzed. This provides a significant memory savings, but exacts a cost of recomputing state information. We have shown that this tradeoff is advantageous when system failure occurs after a small number of component failures. In addition the approach is easily parallelized, either on a dedicated multiprocessor or on an ordinary network of workstations. An important part of our method is to use a minimum of specialized syntax to describe a framework for a model's transition behavior, and to let a modeler use the full resources of the C programming language to describe the details of that behavior.

We also investigate the integration of SURE bounds and Monte Carlo simulation based on importance sampling. We find that the approach produces accurate results using as few as 10,000 replications on models with two orders of magnitude more states. Consequently, on large models the simulation-based analysis executes more quickly. Furthermore, we observe that SURE-based Monte

Carlo estimation has desirable variance reduction properties. Finally, simulation-based analysis admits solution of problems that are too large for exact analysis, and admits easy exploitation of parallelism by simulating independent replications in parallel.

All of the methods described are encorporated in a tool called ASSURE. From a single user interface, ASSURE provides exact analysis or simulation-based analysis, serial execution or parallel execution. Empirical studies of large models solved with ASSURE show that the methods we describe are effective in accelerating the solution time of large complex problems.

Our results show the promise of attacking large reliability problems by path analysis. Further work may be directed towards generalizing the SURE bounds to include non-homogeneous failure rates, and to sharpen confidence intervals with more advance importance sampling schemes.

Distribution

ASSURE is available by request. Contact the first author at `nicol@cs.wm.edu` .

References

[1] M. Boyd and S. Bavusco. Simulation modeling for long duration spacecraft control systems. In *Proceedings of 1993 Annual Reliability and Maintainability Symposium.* IEEE Press, Jan. 1993.

[2] R. Butler. An abstract language for specifying Markov reliability models. *IEEE Trans. on Reliability*, R-35(5):595–601, December 1986.

[3] R.W. Butler and A.L. White. SURE reliability analysis. Technical Report Tech. Paper 2764, NASA Langely Research Center, March 1989.

[4] A. Conway and A. Goyal. Monte Carlo simulation of computer system availability/reliability models. In *Proceedings of the* 17^{th} *International Symposium on Fault-Tolerant Computing.* CS Press, July 1987.

[5] J. Dugan, K. Trivedi, M. Smotherman, and R. Geist. The hybrid automated reliability predictor. *AIAA Journal of Guidance, Control and Dynamics*, 9(3):319–331, May-June 1986.

[6] J.B. Dugan. Fault trees and imperfect coverage. *IEEE Transactions on Reliability*, R-38, June 1989.

[7] A. Goyal et al. The system availability estimator. In *Proceedings of the* 16^{th} *Int'l Symposium on Fault-Tolerant Computing*, pages 84–89. CS Press, 1986.

[8] R. Geist and M. Smotherman. Ultrahigh reliability estimates through simulation. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 350–355. IEEE Reliability Society, January 1989.

[9] K.K. Goswami and R.K. Iyer. DEPEND:A design environment for prediction and evaluation of system dependability. In *Proceedings of the 9^{th} Digital Avionics Systems Conference*. IEEE Press, Oct. 1990.

[10] A. Goyal, P. Shahabuddin, P. Heidelberger, V. Nicola, and P. Glynn. A unified framework for simulating Markovian models of highly dependable systems. *IEEE Trans. Computers*, 41(1):36–51, January 1992.

[11] S. Johnson. The ASSIST language user's manual. Technical Report NASA Technical Memorandum 87735, NASA Langley Research Center, 1986.

[12] J.H. Lala. Advanced information processing system (AIPS) - based fault tolerant avionics architecture for launch vehicles,. In 9^{th} *Digital Avionics Systems Conference*, pages 125–132, October 1990.

[13] E. Lewis and F. Bohm. Monte Carlo simulation of Markov unreliabilty models. *Nuclear Engineering and Design*, 77:49–62, 1984.

[14] D. Nicol. Communication efficient global load balancing. In *Proceedings of the 1992 Scalable High Performance Computing Conference*. IEEE Press, April 1992.

[15] D. Nicol, D. Palumbo, and A. Rikfin. REST:A parallelized system of reliability estimation. In *Proceedings of the Annual Reliability and Maintainability Symposium*. IEEE Reliability Society, 1993.

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

[17] R. Sahner and K. Trivedi. Reliability modeling using SHARPE. *IEEE Transactions on Reliability*, R-36:186–193, June 1987.

[18] T. Sharma and I. Bazovsky. Reliability analysis of large systems by Markov techniques. In *Proceedings of 1993 Annual Reliability and Maintainability Symposium*. IEEE Press, Jan. 1993.

[19] A.L. White and D.L. Palumbo. State reduction for semi-markov reliability models. In *Proceedings of 1990 Annual Reliability and Maintainability Symposium*. IEEE Press, Jan. 1990.

ASSURE User's Guide

Subhendu Das *David Nicol*
College of William and Mary

February 17, 1993

1 Introduction

The ASSURE program was developed to rapidly analyze the reliability of avionics control systems, on a parallel computer architecture. As its name suggests, ASSURE combines the functions of the existing SURE [2] and ASSIST [3] tools developed at the NASA Langley Research Center by Butler, Johnson, and White. ASSURE's analysis is identical to that of SURE. ASSURE accepts reliability models developed using the ASSIST grammar. In addition, ASSURE extends ASSIST syntax by allowing one to directly refer to functions written in C within the model description. The principle utility of this extension is to permit the *calculation* (rather than an ASSIST-style *specification*) of modified state variables, initial state values, and static read-only data-structures. In addition, ASSURE permits Monte-Carlo based analysis, and automatically parallelizes either SURE-based analysis or Monte Carlo analysis.

ASSURE differs algorithmically from ASSIST and SURE, in that it simultaneously creates portions of the state space (like ASSIST) analyzes it (like SURE) but then discards the memory used to represent that portion of the state space. ASSURE thereby achieves considerable savings in memory usage; for models that are moderately large and larger ASSURE runs significantly faster than ASSIST/SURE. However, ASSURE's functionality is more limited than SURE and ASSIST—it will report the reliability bounds, but little less. ASSURE's proper role in the ASSIST/SURE/ASSURE tool suite is as a fast computational engine. Models are most easily developed using ASSIST and SURE; ASSURE can then be used to quickly grind out relability bounds as problem parameters (e.g. pruning level) are varied. The Monte Carlo analysis option extends this capability to systems that are even larger than ASSURE's exact analysis capability can attack.

ASSURE runs only under the Unix operating system. It assumes the availability of the standard Unix utilities *lex, yacc, sed, grep, awk, rsh* the *csh* shell, and *cc.*

This documentation has several parts. Section 2 describes how one sets up an ASSURE system. Section 3 explains how to run ASSURE. Section 4 describes the C language extensions to ASSIST. Section 5 illustrates ASSURE's use through an example of a complex system. Sectionsec:Commands/Constraints summarizes the command scripts through which a user invokes ASSURE, enumerates constraints ASSURE places

on the modeler, and explains error messages. Finally, Section 7 describes the format of a user written routine for modeling trimming.

2 Setting up ASSURE

ASSURE is distributed as a file called `AssureVX.tar.Z`, where X is the current version number (presently 5). To build ASSURE, create a base directory (assumed here to be called `assure`) and a subdirectory `assure/src` into which `AssureVX.tar.Z` is placed. Then uncompress and untar the file, e.g.,

```
% uncompress AssureV5.tar.Z
% tar -xf AssureV5.tar
```

Next, execute a build script `setupassure` to build the ASSIST parsers and pre-compile the problem independent code.

```
% setupassure
```

Various steps in the setting up process are reported to the screen. One of the results of executing *setupassure* is that a new subdirectory of `assure` is created, called `obj`. All of the compilation and file manipulation that occurs in the course of using ASSURE happens in `obj` and a further subdirectory `obj/working`; the problem independent object code is also kept in `obj`. The contents of `src` may therefore be compressed or archived. Finally, move up to the base directory `assure`, and we're ready to go.

3 Running ASSURE

ASSURE will evaluate any ASSIST file subject to constraints identified in Section 6. From the `assure` directory, one executes

```
% runassure filename
```

where `filename` is the name of an ASSIST model file. This file must reside in the base `assure` directory.

`runassure` requires additional option flags when the C extensions are used. As will be described in Section 4.1, the C source code for initialization must be placed in one file; to signal the inclusion of this file and its role as initialization code one includes a `-i filename` on the `runassure` command line. Likewise, the source for testing death-state and transition conditions must be placed in a distinct file and flagged with a `-d`, and the source for modifying a state-vector following a transition is in a distinct file and is flagged with a `-t`. Each type of extension is optional. Any global data and/or data structure definitions, and macro definitions that do not explicitly reference the ASSIST model system state variables ought to be placed at the front of the file flagged with `-i`.

Four other runtime options that are always legal are given below.

`-upper` This option converts all alphabetic characters in the ASSIST model file (but not the extension files) to uppercase. This ensures that the ASSIST keywords are all in uppercase, which ASSURE requires.

`-dbx` Setting `-dbx` causes all ASSURE source code to be copied into subdirectory `assure/obj/working`, where it is compiled into executable file `assure-run`, and executed. The option is useful while debugging, as one can invoke a debugging (such as dbx) on the executable.

`-machines file` This option indicates that this execution ought to be parallelized. `file` is the name of a file in the `assure` directory that contains the network file names (recognizable by `rsh`) of machines to use in the parallelization. All machines must share an NFS file system in which `assure` is built—every machine will be given the same path-name of where to locate `assure-run`, and every machine will execute the same executable image.

`-time t` This option gives a mission time, overridding the mission time declared in the ASSIST file.

Four additional options can be invoked only when using SURE analysis (as opposed to Monte Carlo analysis, to be described).

`-userprune n` This option specifies that the user has supplied a routine to aid in pruning. The value of `n` gives the smallest "level" (number of transitions from initial state) at which one ought to begin to engage this routine. `n` ought to be set to a level 2-3 less than the level at which you desire pruning to be conducted. The specifics are described in Section 7.

`-fop n` This option declares that the system ought only to test whether any death-states are encountered in any sequence of `n` component failures.

`-prune p` This option sets ASSURE's pruning level to `p`, overriding any pruning level declared in the ASSIST model.

`-nomono` ASSURE's pruning mechanism assumes that the sum of all slow component failure rates never exceeds that of the sum out of the initial state. If this assumption does not **not** hold, the user ought to include `-nomono` on the command line, causing a more conservative pruning method to be used.

`-monoto n` This option declares that the sum of all slow component failures out of a state does not increase along any path of `n` or fewer transitions from the initial state.

ASSURE supports both SURE based and Monte Carlo based analysis. To select Monte Carlo analysis one specifies a command line argument `-r n` where `n` gives the total number of replications desired. For example, `-r 10000` serves as a flag to link into the Monte Carlo analysis engine, and perform 10000 replications. Some other command line options are available when doing Monte Carlo analysis.

`-s seed` This option lets you set the random number generator seed as desired. Two runs of the same model with the same specified seed will yield the exact same sample paths.

`-trace n` This option causes trace information to be printed every `n` replications.

`-dc freq` This is option is used when death-state checking is very expensive, and the number of failures tolerated before failure is large. The value of `freq` ought to be set to an estimated average number of component failures the system will tolerate.

`-pruneif` This option declares that one wishes PRUNEIF (and PRUNIF) states to be counted as death-states. They are otherwise ignored (since their intented use is to prune the state-generation process in ASSIST).

`-thrs p` This option relates to ASSURE's default importance sampling scheme. When choosing between the fast class or the slow class, the slow class is chosen with probability `p`. The default value is $p = 0.5$.

`-grab n` This option has meaning when performing parallel Monte Carlo analysis. Load balancing is "self-scheduling", meaning that a processor gets more workload (replications) for itself whenever it has exhausted its present supply. `-grab n` indicates that when it grabs workload, it ought to grab `n` replications. `n` ought to be set so that a processor executes at least several seconds for every group of replications "grabbed". The default setting is the total number of replications divided by the number of processors.

4 Extended ASSIST

ASSURE extends the ASSIST syntax by permitting some of its statements to be replaced by C-functions. These C-functions are defined by the user in auxiliary files which depend on their usage. Any such C-function may refer to any ASSIST model state vector variable; it should interpret each state component as having type *int*. In this section we describe how these extensions are properly used.

4.0.1 TRANTO Statement

TRANTO statements can call C-functions to algorithmically *compute* state vector modifications following a transition. All such functions have to be stored together in a file which is included on the `runassure` command line with the `-t` extension. The syntax of calling a function in the TRANTO statement is given below.

TRANTO (function_name(single_parameter)) BY rate;
Thus, one may replace a sequence of ASSIST state vector modifiers with a call to a C function which performs the same task. This function may include a single, optional, parameter. To illustrate, consider the following example.
Normal TRANTO statement:

```
TRANTO NCF = NCF+1, PE[4+I]=0, PE[I]=0, FAILURE = 1,
       spares[i]=spares[i]+1, ns=ns-1 BY C*LAMP+LAMT;
```

TRANTO statement with C-function:

```
TRANTO (C_func(I)) BY C*LAMP+LAMT;
```

C-function in the `-t` flagged file will be the following:

```
C_func(J)
int J;
{

   NCF = NCF+1;
   PE[4+I] = PE[4+I] + 2;
   PE[I] = PE[I] - 1;
   spares[I] = spares[I]+1;
   ns = ns-1;
}
```

It is important to note that the function call is surrounded by parenthesis, and that the function contains no more than one parameter. Note the usage of ASSIST state variables as simple C variables. The internal ASSURE system treats these variables as having type *int*. This fact should be borne in mind while writing the C functions.

It is even possible to pass pointers to system state variables to subroutines which do not explicitly reference state variables. However, caution is advised—it must never be forgotten that the components have type *int*; furthermore, the internal ASSURE mechanism uses the standard C convention of starting arrays at index 0, rather than 1. Also, the internal ASSURE mechanism expects references to state variable arrays to always specify an array element. The implications of this are illustrated by example. Consider a function that sums the first three elements of a state vector array SE and stores it in the fourth:

```
SumVer1()
{
 int i;
 SE[4] = 0;
 for(i=1; i<=3; i++) SE[4] += SE[i];
}
```

We might write a commonly used subroutine `SumSub()` to sum the first $j-1$ elements of a vector and store it in the jth. This subroutine would be placed in the initialization file.

```
SumSub(j,V)
  int j;
  int *V;
```

```
{
 int i;
 V[j] = 0;
 for(i=0; i<j; i++) V[j] += V[i];
}
```

We could then replace SumVer1 with SumVer2 which calls SumSub:

```
SumVer2()
{
 SumSub(4,&SE[1]);
}
```

The first important feature of this example is that within the file which permits explicit reference to state variables, array index of the first *logical* SE element is 1, while in the ordinary C function the array begins at index 0. This is explained by understanding that the ASSURE process for transforming references to state variables into references to internal C variables automatically subtracts one from all state vector array indices. The second important feature of this example is that while no explicit declaration of SE as an array of char is needed within the file containing SumVer1() and SumVer2(), the SumSub() subroutine must recognize the array as being of type int.

4.0.2 DEATHIF and Condition Statements

Similar to the TRANTO statements, C-functions can be used to flag death-state conditions, and transition conditions. The C-functions called must return a value of type integer, to indicate the outcome of their testing. The normal C convention for Boolean interpretation of integers is followed; 0 ⇒ FALSE, otherwise TRUE. All such functions are contained in a common file, which is flagged on the runassure command line with a *-d*. The syntax for using such functions is given below. Note again the need for parenthesis around the function calls, and the limit of one functional parameter.

DEATHIF (function_name(single_parameter));

Example:

Normal DEATHIF statement:

```
DEATHIF ((seca<3 and pri<2) or vlv1<2) and
        ((secb<3 and (pri=2 or pri=0)) or vlv2<2) or
        (seca=1 and pri<2) or (secb=1 and (pri=2 or pri=0)) or
          vlv1=0 or vlv2=0;
```

DEATHIF statement with C-function:

```
DEATHIF (Death_Cond());
```

C-function in DEATHIF_and_COND_EXPR_file will be the following:

```
int Death_Cond()
{

  if (((seca<3 and pri<2) or vlv1<2) and
        ((secb<3 and (pri=2 or pri=0)) or vlv2<2) or
        (seca=1 and pri<2) or (secb=1 and (pri=2 or pri=0)) or
          vlv1=0 or vlv2=0)
     return(1);
  else
     return(0);
}
```

IF (function_name(single_parameter))

Example:

Normal Condition Expression statement:

```
IF PE[1] + PE[2] + PE[3] + PE[4] > 0
  TRANTO .......
```

Condition Expression statement with C-function:

```
IF (Cond_func())
  TRANTO .......
```

C-function in DEATHIF_and_COND_EXPR_file will be the following:

```
int Cond_func()
{

  if PE[1] + PE[2] + PE[3] + PE[4] > 0
     return(1);
  else
     return(0);
}
```

4.0.3 Initialization and Global Data Structures

It is sometimes desirable to engage in certain initialization activities. For example, the start state given in the ASSIST file can be modified to reflect components that are most easily computed, rather than figured out by hand and included in the ASSIST START statement. Any such initialization to be done has to be a consequence of executing a procedure whose name must be `Initialize()`. The function, and any others which do not explicitly reference ASSIST state vector components are gathered together in a single file which is flagged on the `runassure` command line with a `-i`. Any file flagged by `-i` must include a function named `Initialize()`, even if that function is null. It is also useful to define certain global data structures to be used by the various C-functions. These definitions should be placed at the head of the file flagged by `-i`.

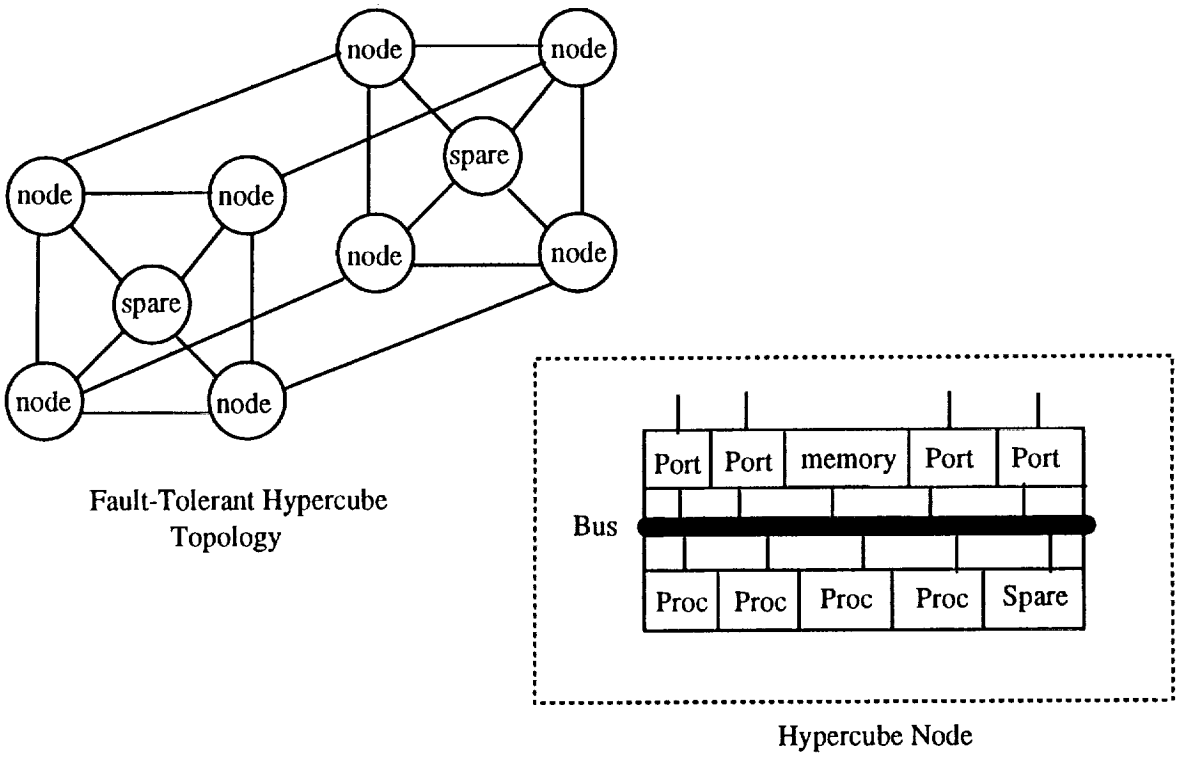


Figure 1: Example: Fault-tolerant hypercube architecture

5 Example

We now illustrate the use of ASSURE using an example. The problem is based upon the fault-tolerant architecture described in [1], illustrated in Figure 1. The basic architecture is that of a three-dimensional hypercube of fault-tolerant processing nodes, with one spare processing node for each of two two-dimensional subcubes. For the purposes of this example, the spares are assumed to be cold. Each processing node is comprised of (i) four communication ports, (ii) four processors, (iii) one spare processor (assumed to be hot), (iv) one shared memory, and (v) one communication bus. A node is considered to have failed if fewer than four processors are operative, or if fewer than three links are operative, or if either the memory or bus have failed. Upon failure of a processing node, the appropriate spare is brought in, if possible, after a recovery delay. A *subcube* is considered to have failed if more than one if its nodes has failed. Finally, the *system* is considered to have failed if either subcube has failed, or if two operative nodes in a non-recovery state are unable to communicate.

5.1 ASSURE Files

The entire ASSURE model of this problem is given at the back of this document. It is comprised of files `cb.ast`, `cb.i`, `cb.t`, `cb.d`. These files are also found in subdirectory `src/problems/space-cube` of the ASSURE distribution.

Let us first examine `cb.ast`. Here we see that the mission is one year (in hours).

Other elements of the preamble declare the various transition rates (in failures/hr), and declare three statistics variables. The ten processing nodes are indexed from 1 to 10, with the two spares being numbers 9 and 10. Most state variables are indexed by the identity of a processing node. `NODE[i]` describes the state of a node, as having attributes GOOD, BAD, INUSE, and DORMANT. `MEM_BUS[i]` is 1 if both the memory and bus of node `i` are operative. `PROC[i]` gives the number of operative processors (plus spare) in the node, and `PORT[i]` gives the number of operative communication ports. There are twenty links in the system, indexed from 1 to 20. Array `LINK` gives the state (GOOD,BAD) of each. Finally, `SUB[1]` gives the state of one subcube (nodes 1-4 and 9) while `SUB[2]` gives the state of the other.

The START statement gives default initial state variable values. These may be changed (and some are) by the initialization routine `Initialize()`.

The two DEATHIF statements express the two high level requirements for failure—one of the subcubes has failed, or we are unable to establish connectivity. The first of these is tested directly using ASSIST syntax. The second of these calls a C function to determine the connectivity. Later we will see how individual component errors may percolate up and have system failure be reflected in a `SUB` variable.

The TRANTO loop over `K=1,10` identifies processors, memories, and buses that may fail. Since either a memory or bus failure will fail a node, we have combined the effects of the two elements into one. The loop over `K=9,10` identifies components of on-line spare nodes that may be failed.

The loop over `K=1,20` identifies links that may fail. For simplicity we have combined the failure of a link and either of its end-point ports.

The last set of transition rules are for recovery transitions, flagged by the keyword FAST. The first operative node to fail in a subcube initiates a subcube recovery, where the cold spare is brought in. To reflect this transition, the appropriate component of `SUB` is set to REPAIRING. Completion of this transition restores the `SUB` state to GOOD, and brings the spare node's components on-line. Before examining the details of the various C routines that are referenced, consider the overall structure. Each component or subsystem in this model has an "Effect" routine. Calling an Effect routine for an elemental component is a signal to fail the component. Calling an Effect routine for a subsystem (like NODE, or SUB) is a signal that at least one of the subsystem's constituent parts has changed, and so the subsystem ought to evaluate its own state. Thus, the ASSURE loop calls a component's Effect routine to signal a failure, e.g., `ProcEffect`. This routine decrements the number of active processors, and then calls `Nodeffect`, the Effect routine for its subsystem NODE. `NodeEffect` determines whether the node is still alive. If not, it signals node failure to the next higher subsystem containing it, by calling `SubCubeEffect`. This routine determines whether the node failure brings down the subcube, or whether there exists a spare and the spare can be brought in to replace the failure. If the spare can be brought in, the appropriate `SUB` component is modified to value RECOVERING. It is otherwise set to FAILED, a condition that will be discovered with a DEATHIF check. Thus, we use ASSURE to construct a hierarchical layer of Effect functions that determine local subsystem state conditions, and signal higher levels when events occur that may affect them.

Now consider file `cb.i`. At the beginning we find macro definitions, structure definitions, and global variables. Next there is routine `Initialize`. Note that it selects an initial set of node and links to be in-use, thereby overriding the initial values given by the START statement. It also calls a routine `maplinks()` that generates some global data structures that describe the network topology.

File `cb.d` contains routine `ConnectionFailure()`, which determines whether a network that is not under repair is still connected. Note the early exit if the states of either `SUB` variable reflect an ongoing transition. Recall that `IsUnderRepair` is a macro defined in `cb.i`. The routine works by building a spanning-tree rooted at any `NODE` that is in use. The spanning-tree is built using a recursive depth-first traversal that "marks" every node it touches. Thus, the network is disconnected if and only if there exists a node that is not reachable from the root though the spanning-tree.

`cb.d` also contains routine `Measure()`, which we use with the simulation option. This routine measures the number of processors, links, and memory/bus components that are not failed at the point a death-state is entered. We also find routine `TrimAnalysis()`, which is used as part of the user-assisted trimming option discussed early. `TrimAnalysis()` loops over operational nodes. It first analyses the criticality of the node's subcube—a critical subcube is one that cannot tolerate a node failure. Next it looks at the number of working processors in the node. If there is a full complement, then one of them can fail, at any time, and not bring the system down. Hence, there is a *safe* transition (see [4], included with the ASSURE distribution) with rate `5*PF` (recall that `PF` is the processor failure rate). If the node has only four processors, a failure by one will drop the node, and may drop the system—depending on whether the subcube is critical. Thus the next processor failure is *conditionally safe* if the subcube is critical, and is *unsafe* otherwise. Similar logic applies for the node's `MEM_BUS` component.

File `cb.t` contains the Effect functions. As previously described, Effect functions for elemental components `PROC`, `LINK`, and `MEM_BUS` simply mark the component as having failed, propagate the effects locally (e.g., `LinkEffect` alters port counts in affected nodes), and then call the Effect function for `NODE`. On being called `NodeEffect()` analyzes all components of the `NODE` state. Observe that `NodeEffect()` does not know the reason it is called. This is deliberate. Each Effect function ought to be completely self-contained, and free from any pre-determined assumptions about the state of the system. If this policy is adhered to in every Effect module, the logic of model will be correct. The logic will otherwise depend on assumptions concerning the order of Effect function evaluation. If `NodeEffect` determines that the node has failed, then all components touching the node are treated as though they have failed. This is an entirely practical matter, relating to the size of the state-space ASSURE must explore. Since later failures of components in an already failed node cannot further degrade the system state, there is no point in modeling those failures. Finally, `SubCubeEffect` is called. This routine examines the state of the subcube and determines whether repair is possible. If it is the subcube state is set to "repairing" mode, otherwise the subcube state is set to failed (which will be detected by the DEATHIF conditions).

5.2 Some Sample Runs

Now let us examine ASSURE's output on this problem. Figure 2 illustrates the command line and resulting information from using SURE-based analysis on a single workstation. First one sees that as ASSURE transforms the model into C code, it reports on intermediate steps. This takes about a minute. The statement "This is ASSURE ..." is printed just prior to execution of the ASSURE model. The first line of output reports that 801,842 states were generated in the course of this run, and that 778,882 paths were analyzed. Remember that the node count does not give the number of *unique* states, ASSURE regenerates states as needed along a path. The second and third lines of output report that 138,012 death-states were discovered; `LBOUND` and `UBOUND` give the sum of upper and lower bounds on the probabilities of reaching these states within the mission time. The `P-SUBTOTAL` numbers relate to states pruned by ASSIST PRUNEIF statements, of which there are none in this model The `TOTAL` bounds are the sum of the `D-SUBTOTAL` and `P-SUBTOTAL` bounds. The sum of pruned states probabilities gives the sum of upper bounds on the paths that are pruned. To construct a conservative upper bound on the probability of failure one adds this sum to the `TOTAL UBOUND`. The execution is prefaced with the Unix "time" command, which explains the cryptic last line. The first three numbers relate to running time: this run required 1641.32 "user" seconds of CPU time, 2.13 "system" seconds, and an elapsed wallclock time of 27 minutes, 51 seconds.

Now let us consider a parallel execution of ASSURE. We construct a file, `mf.10`, with names of 10 workstations that all have the same NFS'd view of the directory in which ASSURE is built. As shown with the `more` command in Figure 3, the William & Mary workstations are named after states. Then we add `-machines mf.10` to the command line. Unlike the serial ASSURE command, the parallel version returns control to the command line as soon as the execution is initiated. In order to monitor the progress of the computation, we invoke the script `Peek`, whose ultimate results are shown. `Peek` displays the time at which the run was initiated, and for every workstation shows the total number of nodes processed and its termination time. These times are taken from time-stamps on files, and hence have a resolution of about 1 minute. Following this report `Peek` gives a summary similar to that in the serial case. Happily we observe that the parallel version finds precisely the same result, and achieves a significant speedup (a 3-4 parallel minute run versus a 28 minute serial run).

Now let's do a serial Monte Carlo analysis. As shown in Figure 4, we remove the `-userprune` option (since no pruning is done during simulation), and include `-r 10000` and `-dc 10`. The first of these signals our intention that Monte Carlo analysis be done, with 10,000 replications, the second invokes the death-state checking optimization. The output is understood as follows. Under the Monte Carlo option the upper bound and the lower bound are considered to be "statistics", just as are any declared SAMPLE variables. A confidence interval is constructed for each statistic. The output gives, for every statistic, the sample mean (called "point estimate") and sample standard deviation, the 95% confidence interval, and the "relative error" which is measured as the width of the confidence interval divided by the point estimate. The smaller the

```
%runassure cb.ast -i cb.i -t cb.t -d cb.d -userprune 3
Translation of assist file to C code
.'bd1.c:
bdx.c:
Compilation of problem specific code
Linking:
This is ASSURE Version 5.0 (SURE Analysis)
 the no. of nodes 801842, no. of paths 778882
D-SUBTOTAL :LBOUND:7.079588e-04  UBOUND:7.080437e-04
The no. of deathstates are : 138012

P-SUBTOTAL  :LBOUND:0.000000e+00  UBOUND:0.000000e+00


TOTAL :LBOUND:7.079588e-04  UBOUND:7.080437e-04

The sum of pruned states probabilities < 7.255260e-05

The no of paths pruned at level 1.000000e-09 is 640870

The avg # transitions for a path is (a:4.669932e+00,s:4.652597e+00)
1641.430u 2.130s 27:51.73 98.3% 0+198k 0+0io 0pf+0w
```

Figure 2: Example of serial SURE based analysis run

```
%more mf.10
nc ct md ri ny ga de mn in il
%runassure cb.ast -i cb.i -t cb.t -d cb.d -userprune 3 -machines mf.10
Translation of assist file to C code
.''bd1.c:
bdx.c:
Compilation of problem specific code
This is ASSURE Version 5.0 (Parallel SURE Analysis)
   Execute script Peek to monitor progress of analysis
%Peek
Run began at Feb 17 10:12
        Processor 0 did 79284 nodes by time Feb 17 10:14
        Processor 1 did 91481 nodes by time Feb 17 10:14
        Processor 2 did 73436 nodes by time Feb 17 10:15
        Processor 3 did 79872 nodes by time Feb 17 10:14
        Processor 4 did 73436 nodes by time Feb 17 10:14
        Processor 5 did 81815 nodes by time Feb 17 10:14
        Processor 6 did 79284 nodes by time Feb 17 10:14
        Processor 7 did 91481 nodes by time Feb 17 10:15
        Processor 8 did 72164 nodes by time Feb 17 10:15
        Processor 9 did 79589 nodes by time Feb 17 10:15


With 10 of 10 processors reporting.....
the no. of nodes 801842
D-SUBTOTAL :LBOUND:7.079588e-04  UBOUND:7.080437e-04
The no. of deathstates are : 138012

P-SUBTOTAL  :LBOUND:0.000000e+00  UBOUND:0.000000e+00


TOTAL :LBOUND:7.079588e-04  UBOUND:7.080437e-04

The sum of pruned states probabilities < 7.255260e-05

The no of paths pruned at level 1.000000e-09 is 640870
```

Figure 3: Example of parallel SURE based analysis run

relative error, the tighter the confidence interval is. We see then that the reported confidence intervals for upper and lower bounds statistics `UB` and `LB` do indeed contain the bounds given by the SURE analysis. Furthermore, the Monte Carlo option's running time is somewhat less— 5 minutes versus 28 minutes.

We invoke the parallel Monte Carlo option just as we did the parallel SURE option, by simply adding a machine-list file name to the command-line. This is shown in Figure 5. For good measure we have boosted the number of replications to 100,000. As with the parallel SURE option, we monitor the computation by executing script `Peek`.

6 Commands, Constraints, and Errors

6.1 Command Scripts

The user's interface to ASSURE is through command scripts.

`setupassure` This is executed from the `assure/src` subdirectory. It either creates, or overwrites directory `assure/obj`, building ASSURE's parsers and problem independent object code. This need only be executed once.

`runassure` This script is called from the base directory, `assure`. Its first argument must always be an ASSIST model file name, which must also reside in the base directory. Other extensions are possible, as described earlier in the document. The effect of calling `runassure` is to parse the ASSIST file and create problem-specific C language files which are compiled, and linked with the problem-independent object files. The resulting file is executed to solve the problem.

`rerunassure` This script provides a short-cut to re-run a model with a different set of problem parameters, by avoiding the translation and compilation step. It is necessary that the "re-run" be of the same type as the previous run, e.g., both SURE-based, or both Monte-Carlo based. Options that affect problem code compilation— notably `-is` and `-userprune n`—must be identical between runs. `rerunassure` is called from the base `assure` directory.

`KillNetRun` This option is executed to prematurely kill a distributed ASSURE run, either SURE-based or Monte Carlo based. Whenever a workstation begins its execution of a distributed ASSURE run, it creates a shell script for killing itself via the Unix `kill` command. `KillNetRun` simply remotely executes each one of these scripts. It may happen that `KillNetRun` attempts to kill a workstation that has already completed, in which case an error message is printed. These may be ignored. `KillNetRun` is called from the basis `assure` directory.

`Peek` This script is called to monitor the progress of a distributed ASSURE computation. It too is called from the basis `assure` directory. The user infers the completion or non-completion of the computation by examination of the `Peek` output. For a SURE run, the computation is completed when `Peek` reveals that all processors

```
%runassure cb.ast -i cb.i -t cb.t -d cb.d -r 10000 -dc 10
Translation of assist file to C code
.''bd1.c:
bdx.c:
Compilation of problem specific code
This is ASSURE Version 5.0 (Simulation Analysis)
--- Fast/Slow Class:Prop. Transition Sampling (Pr{slow} = 0.500000) ----
 %%%%%%%% RESULTS FOLLOWING 10000 REPLICATIONS %%%%%%%%
-----------------  Statistic UB -------------------
       point est. = 6.200053e-04
       std dev.   = 6.802262e-03
95% conf. int. (4.866810e-04, 7.533297e-04)
       rel. err.  = 3.539602e-01
-----------------  Statistic LB -------------------
       point est. = 6.199230e-04
       std dev.   = 6.801605e-03
95% conf. int. (4.866116e-04, 7.532345e-04)
       rel. err.  = 3.539707e-01
-----------------  Statistic LIVEPROC -------------------
       point est. = 7.279000e+00
       std dev.   = 1.037249e+01
95% conf. int. (7.075699e+00, 7.482301e+00)
       rel. err.  = 5.434179e-02
-----------------  Statistic LIVEGUTS -------------------
       point est. = 7.279000e+00
       std dev.   = 1.037249e+01
95% conf. int. (7.075699e+00, 7.482301e+00)
       rel. err.  = 5.434179e-02
-----------------  Statistic LIVELINK -------------------
       point est. = 1.186240e+01
       std dev.   = 1.671584e+01
95% conf. int. (1.153477e+01, 1.219003e+01)
       rel. err.  = 5.375385e-02
340.4u 1.0s 5:46 98% 0+364k 0+0io 0pf+0w
```

Figure 4: Example of serial Monte Carlo based analysis run

```
%runassure cb.ast -i cb.i -t cb.t -d cb.d -machines mf.10 -r 100000 -grab 1000
Translation of assist file to C code
.''bd1.c:
bdx.c:
Compilation of problem specific code
This is ASSURE Version 5.0 (Parallel Simulation Analysis)
   Execute script Peek to monitor progress of analysis
%Peek
Statistics from 100000 replications total
Run began at Feb 17 10:59
        Processor 0 has 10946 reps at time Feb 17 11:04
        Processor 1 has 12002 reps at time Feb 17 11:04
        Processor 2 has 6003 reps at time Feb 17 11:04
        Processor 3 has 12004 reps at time Feb 17 11:04
        Processor 4 has 12005 reps at time Feb 17 11:04
        Processor 5 has 11006 reps at time Feb 17 11:04
        Processor 6 has 12007 reps at time Feb 17 11:04
        Processor 7 has 8008 reps at time Feb 17 11:04
        Processor 8 has 8009 reps at time Feb 17 11:04
        Processor 9 has 8010 reps at time Feb 17 11:04
----------------  Statistic UB -------------------
       point est. = 6.866804e-04
       std dev.   = 8.211339e-03
95% conf. int. (6.357860e-04, 7.375748e-04)
       rel. err.  = 1.380047e-01
----------------  Statistic LB -------------------
       point est. = 6.865932e-04
       std dev.   = 8.210591e-03
95% conf. int. (6.357034e-04, 7.374829e-04)
       rel. err.  = 1.380093e-01
----------------  Statistic LIVEPROC -------------------
       point est. = 7.451200e+00
       std dev.   = 1.039726e+01
95% conf. int. (7.386757e+00, 7.515643e+00)
       rel. err.  = 1.714900e-02
----------------  Statistic LIVEGUTS -------------------
       point est. = 7.451200e+00
       std dev.   = 1.039726e+01
95% conf. int. (7.386757e+00, 7.515643e+00)
       rel. err.  = 1.714900e-02
----------------  Statistic LIVELINK -------------------
       point est. = 1.212298e+01
       std dev.   = 1.671392e+01
95% conf. int. (1.201939e+01, 1.222657e+01)
       rel. err.  = 1.694570e-02
```


Figure 5: Example of parallel Monte Carlo based analysis run

have completed. In contrast, the Monte Carlo version provides intermediate statistics, even if the computation has not completed. The computation is known to be completed when the number of replications reported equals the number requested. There is no need to call `KillNetRun` if the distributed computation is shown to have completed normally.

6.2 Constraints

There are certain important assumptions that have been made about the input ASSIST file. These have to be followed so that it can be parsed for the translation process.

- The reserved words are in CAPS. ASSIST is treated as case sensitive.
- DEATHIF and PRUNEIF statements have to follow the Start statement.
- If any statement contains a**b then it has to be enclosed in parenthesis.
- Arrays of constants are not permitted; nor are not-system- state variable arrays. Can do it using C.
- Cannot use definitions of the form
  A = 5 TO 50 BY 5;
- The FOR statement has to end with a semicolon.
- The functions should be in C.
- If functions are used in TRANTO statements then it has to be within parenthesis
  IF cond_expr
  TRANTO (function_name()) BY rate;
- If functions are used in DEATHIF statements then it has to be of the following form
  DEATHIF (function_name());
- Mission time is a real number and has to be defined in the ASSIST file.

6.3 Error Messages

The error messages printed by ASSURE are the following:

1. Syntax error in line *line_no* in file *file_name*.
   Reason: Syntax error in the ASSIST code.
2. unable to open file *file_name*.
   Reason: ASSIST file not found.

3. Dynamic memory exhausted
   Reason: No more memory available for carrying on the depth-first search of the state space. Usually an indication that the user model is mismanaging dynamic memory allocation, e.g., not freeing up discarded memory blocks.

4. Various compile errors.

   Syntax errors in a user's model will provoke cc into generating many error messages. It is recommended that the `-dbx` switch *not* be used until the ASSURE model compiles correctly, as the error messages from cc accurately pinpoint the file and line number of the problem. When `-dbx` is set, the compile errors are reported in a file `prob.c`, which is found in `obj/working`. The cause of the error can usually be inferred from the error report on `prob.c`, it is just not as convenient. The reason, in fact, for the `-dbx` switch is that the mechanism used to accurately pin-point compile errors seems to confuse `dbx`. Invoking `-dbx` circumvents the problem, at the cost of more mysterious error messages.

5. Various Command Script Errors

   `runassure` will report that a command line error exists if the command line contains illegal switches. A user may neglect to include a required argument to a switch, in which case `runassure` may or may not complain, depending on the error. In most cases of this type the error is caught, but it may also manifest itself in the assignment of strange unintended values to to problem parameters. Be careful out there.

7 User Assisted Trimming

A user may assist ASSURE with smart trimming, by writing a routine that provides ASSURE with classification of transitions out of the present state. An ability to do this can reduce the number of states analysis by as much as two orders of magnitude.

The user signals ASSURE that this assistance is present with the `-userprune n` switch. This implies that a file with a `-d` extension exists, and contains a function

```
TrimAnalysis(SafeSum, CondSum, UnsafeSum, SumMax,RecovSum )
   float *SafeSum, *CondSum, *UnsafeSum, *SumMax,*RecovSum;
```

ASSURE calls `TrimAnalysis` expecting to have values assigned to the locations pointed to in the argument list. The role of `TrimAnalysis()` is to quickly analyze the ASSIST current state variables, using knowledge of the system structure, to classify transitions out of the current state. These are understood, as follows.

An **unsafe** transition is one that *may* lead to a death-state. It is permitted to classify a transition as **unsafe**, even if the possibility exists that the transition does not cause system failure. The converse is not permitted. `TrimmingBound()` is expected to record the sum of rates of all **unsafe** transitions in the location pointed to by `UnsafeSum`.

Non-**unsafe** transitions are to be classified as **safe** or **conditionally safe**. A **conditionally safe** transition is one that will not cause system failure, if taken immediately.

A **safe** transition is one that may be deferred for one step, and still not cause system failure. Typically instances of **safe** transitions are transitions that cause available hot spares to be immediately switched in. `TrimmingBound()` is expected to record the sum of rates of **conditionally safe** transitions in the location pointed to by `CondSum`, and to record the sum of rates of **safe** transitions in the location pointed to by `SafeSum`. In both cases, upper bounds on these sums suffice. `TrimAnalysis()` is also expected to record a *lower* bound on the sum of rates of possible recovery transitions. in the location pointed to by `RecovSum`.

Finally, `TrimAnalysis()` may *optionally* write into the location pointed to by `SumMax` an upper bound on the sum of slow transition rates out of any state reachable from the present one. ASSURE assumes a value for this, `TrimAnalysis()` may tighten it, if desired.

References

[1] Boyd, M. and Bavusco,S. Simulation Modeling for Long Duration Spacecraft Control Systems, *Proceedings of 1993 Annual Reliability and Maintainability Symposium*, Jan., 1993, pp. 106-113.

[2] Butler,Ricky W.; and White, Alan L.:*SURE Reliability Analysis - Program and Mathematics.* NASA TP-2764, 1988.

[3] Johnson,S.:*The ASSIST language users's manual.* Technical report, NASA Langley Research Center.

[4] Nicol,D. and Palumbo, D. *Reliability Analysis of Large Complex Models Using SURE Bounds.*

```
(* CB.AST *)

TIME= 8760;     (* MISSION TIME = 1 YR *)
PRUNE=1E-9;     (* SURE PRUNE LEVEL *)

(*  FTP AND ITS NETWORK INTERFACES   *)

MF = 3.477E-7;    (* MEMORY FAILURE RATE *)
BF = 1.147E-7;    (* BUS FAILURE RATE *)
PF = 1.99E-6;     (* PROCESSOR FAILURE RATE *)
LF = 1.0E-7;      (* LINK FAILURE RATE *)
RR = 1.0E+3;      (* REPAIR RATE TO SWAP IN SPARE NODE *)

SAMPLE LIVEPROC;
SAMPLE LIVEGUTS;
SAMPLE LIVELINK;

DEAD      = 0;
GOOD      = 1;
INUSE     = 2;
DORMANT   = 4;
REPAIRING = 4;

(* STATE SPACE DEFINITION *)
SPACE = (MEM_BUS:ARRAY[1..10] OF 0..1,
         PROC: ARRAY[1..10] OF 0..5,
         PORT: ARRAY[1..10] OF 0..4,
         LINK: ARRAY[1..20] OF 0..1,
         NODE: ARRAY[1..10] OF 0..1,
         SUB:  ARRAY[1..2] OF 0..2);

START = (10 OF 1,
         10 OF 5, 10 OF 4,
         20 OF 4,  8 OF 3, 2 OF 4,
         2 OF 1);

(* Failure Conditions *)
DEATHIF SUB[1] = DEAD  OR SUB[2] = DEAD;    (* A dead subcube *)
DEATHIF ConnectionFailure();                (* can't connect  *)

(* TRANSITIONS *)
FOR K=1,10;
  (* Fail a working node's parts *)
  IF NODE[K] = INUSE+GOOD  THEN
    IF PROC[K] > 0 TRANTO (ProcEffect(K)) BY PROC[K]*PF;
    IF MEM_BUS[K] = 1 TRANTO (Mem_BusEffect(K)) BY (MF+BF);
 ENDIF;
ENDFOR;

FOR K=1,20;
   IF LINK[K] = INUSE+GOOD TRANTO (LinkEffect(K)) BY LF;
ENDFOR;

(* Repair Transitions *)
IF SUB[1] = REPAIRING TRANTO (SubCubeRepair(1)) BY FAST RR;
IF SUB[2] = REPAIRING TRANTO (SubCubeRepair(2)) BY FAST RR;
```

```
#include <stdio.h>
#define Good 0x1
#define Bad 0x0
#define InUse 0x2
#define Dormant 0x4
#define UnderRepair 0x4
#define SetBad(a) a = 0
#define SetInUse(a) (a = Good|InUse)
#define SetUnderRepair(a) (a = UnderRepair)
#define SetDormant(a) (a = Dormant)
#define IsGood(a)    (a&Good)
#define IsOK(a)      (a&Good)
#define IsInUse(a)   (a&InUse)
#define IsBad(a)     (!a)
#define IsUnderRepair(a) (a&UnderRepair)
#define IsDormant(a) (a&Dormant)

int NumNodes, NumLinks;
int visited[10];
typedef struct conn *CONNPTR;

struct conn{
        int      node, link;
        CONNPTR  next;
        };

struct conn *ptrarray[10];
struct link_end_struct { int end1, end2; } LinkEnds[20];
int NumNodes, NumLinks;


Initialize()
{
int i;
CONNPTR ptr;

        NumNodes = 10; NumLinks = 20;
        maplinks();
        for(i=0; i<8; i++) {
           /* mark first eight nodes as in use */
           SetInUse( NODE[i+i] );
           ptr = ptrarray[i];
           while(ptr) { SetInUse(LINK[ptr->link+1]);
                        ptr = ptr->next;
                      }
        }
        SetDormant( NODE[9] );
        ptr = ptrarray[9-1];
        while(ptr) { SetDormant(LINK[ptr->link+1]);
                        ptr = ptr->next;
                      }
        SetDormant( NODE[10] );
        ptr = ptrarray[10-1];
        while(ptr) { SetDormant(LINK[ptr->link+1]);
                        ptr = ptr->next;
                      }

}

LinkNeighbor(NNum, Neighbor, link, ptr)
 int NNum, Neighbor, *link;
 CONNPTR ptr;
{
int idx = *link;
```

```
CONNPTR tmp;
     if(NNum < Neighbor) {
        LinkEnds[idx].end1 = NNum; LinkEnds[idx].end2 = Neighbor;
        ptr->link = idx; (*link)++;
     }
     else {
       tmp = ptrarray[Neighbor];
       while(tmp->node != NNum) tmp = tmp->next;
       ptr->link = tmp->link;
    }

}


maplinks()
{
CONNPTR ptr,tmp;
int LNum, NNum;

     /* build up basic nodes */
     for(NNum=0, LNum=0; NNum<8; NNum++) {

       ptr = (CONNPTR)malloc(sizeof(struct conn));
       ptr->node = NNum^0x1;
       ptrarray[NNum] = ptr;
       LinkNeighbor(NNum, NNum^0x1, &LNum, ptr);


       ptr->next = (CONNPTR)malloc(sizeof(struct conn));
       ptr = ptr->next;
       ptr->node = NNum^0x2;
       LinkNeighbor(NNum, NNum^0x2, &LNum, ptr);

       LinkEnds[LNum].end1 = NNum;
       LinkEnds[LNum].end2 = (NNum<4?8:9);  /* the spare */
       ptr->next = (CONNPTR)malloc(sizeof(struct conn));
       ptr = ptr->next;
       ptr->node = LinkEnds[LNum].end2;
       ptr->link = LNum++;

       ptr->next = (CONNPTR)malloc(sizeof(struct conn));
       ptr = ptr->next;
       ptr->node = NNum^0x4;
       LinkNeighbor(NNum, NNum^0x4, &LNum, ptr);
       ptr->next = 0;
     }

     /* build the link structures for the spares by hand */
     ptr = (CONNPTR)malloc(sizeof(struct conn));
     ptrarray[8] = ptr;

     ptr->link = 2;
     ptr->node = 0;

     ptr->next = (CONNPTR)malloc(sizeof(struct conn));
     ptr = ptr->next;
     ptr->node =  1;
     ptr->link = 5;

     ptr->next = (CONNPTR)malloc(sizeof(struct conn));
     ptr = ptr->next;
     ptr->node =  2;
     ptr->link = 8;
```

```
    ptr->next = (CONNPTR)malloc(sizeof(struct conn));
    ptr = ptr->next;
    ptr->node =  3;
    ptr->link = 10;
    ptr->next = 0;

    ptr = (CONNPTR)malloc(sizeof(struct conn));
    ptrarray[9] = ptr;

    ptr->link = 14;
    ptr->node = 4;

    ptr->next = (CONNPTR)malloc(sizeof(struct conn));
    ptr = ptr->next;
    ptr->node =  5;
    ptr->link = 16;

    ptr->next = (CONNPTR)malloc(sizeof(struct conn));
    ptr = ptr->next;
    ptr->node =  6;
    ptr->link = 18;

    ptr->next = (CONNPTR)malloc(sizeof(struct conn));
    ptr = ptr->next;
    ptr->node =  7;
    ptr->link = 19;
    ptr->next = 0;

}
```

```
ConnectionFailure()
{
int i,j,base;

        /*------ only check for a connection failure if neither
                 subcube is under repair --------------------*/
        if( (IsUnderRepair(SUB[1])) || (IsUnderRepair(SUB[2]) )) return 0;

        /* find a base for a spanning tree */
        for(i=0;i<10;i++)
          if(IsInUse(NODE[i+1])) break;

        if(i==10) return 1;
        else base = i;

        /* do a depth-first traversal , marking visited nodes */
        for(j=0; j<10; j++) visited[j] = 0;
        dfs(base);

        /* visited[j] is 1 if i can get to j.
             check for violations */

         for(j=0; j<10; j++)
            if( (IsInUse( NODE[j+1])) && (!visited[j]) ) return 1;

         /* touched them all, so we're OK */
         return(0);
}


dfs(nodenum)
 int nodenum;
{
CONNPTR ptr;

        visited[nodenum] = 1;
        ptr = ptrarray[nodenum];
        while(ptr) {
          if( (IsInUse( LINK[ptr->link+1] )) &&
                (!visited[ptr->node])) dfs(ptr->node);
          ptr = ptr->next;
        }
}


Measure()
{
int idx,cnt;

        for(cnt=0,idx=1; idx<=10; idx++) cnt += (int)(PROC[idx] != 0);
        LIVEPROC = cnt;

        for(cnt=0,idx=1; idx<=20; idx++) cnt += (int)(LINK[idx] != 0);
        LIVELINK = cnt;

        for(cnt=0,idx=1; idx<=10; idx++) cnt += (int)(MEM_BUS[idx] != 0);
        LIVEGUTS = cnt;
}

STATETYPE *TmpState;

TrimAnalysis(SafeSum, CondSum, UnsafeSum, SumMax,RecovSum )
 float *SafeSum, *CondSum, *UnsafeSum, *SumMax,*RecovSum;
{
```

```
STATETYPE SaveLink;
float TmpSafeSum, TmpCondSum, TmpUnsafeSum, TmpSumMax, TmpRecovSum;
int idx, Critical, cnt;
STATETYPE *SaveState;

        TmpUnsafeSum = TmpSafeSum = TmpCondSum = TmpSumMax = 0.0;
        TmpRecovSum = 0.0;

        /* determine if the subcubes are critical by looking to see
           if the cube's spare processor is in use
       */

        for(idx=1; idx<=10; idx++) {

                /* don't do anything if this node is bad or dormant */
                if( IsBad( NODE[idx] ) || IsDormant( NODE[idx]) ) continue;

                if(idx==1) Critical =
                   (IsInUse( NODE[9]) || (IsUnderRepair(SUB[1])));
                if(idx==5) Critical =
                   (IsInUse( NODE[10] )||(IsUnderRepair(SUB[2])));
                if(idx==9)
                   (IsInUse( NODE[9] )||(IsUnderRepair(SUB[1])));
                if(idx==10)
                   (IsInUse( NODE[10] )||(IsUnderRepair(SUB[2])));


                /* any PROC component == 5 gives a Unconditional SAFE trans.
                   Any other for an active node gives an UNSAFE transition
                */

                    if( PROC[idx] == 5 ) TmpSafeSum += 5*PF;
                    /* only 4 working processors. Classification
                       depends on whether cube is critical */
                    else  {
                      if(Critical) TmpUnsafeSum += (4*PF);
                      else TmpCondSum += (4*PF);
                    }

                /* any MEM_BUS transition from a critical subcube is unsafe */
                if(Critical) TmpUnsafeSum += (BF+MF);
                else  TmpCondSum += (BF+MF);

        }

        /* examine each link. Any link that disconnects the network
           of InUse nodes is UnSafe.  Any other link is Conditionally
           Safe.  Play some dirty games so that we can call
           CHECK_CONNECT.
        */

        /* fetch state vector space if needed */
        if(!TmpState) TmpState =
                (STATETYPE *)malloc(sizeof(STATETYPE)*num_states);

        /* Save pointer to current state, and copy that state */
        SaveState = atnodeptr->num;
        atnodeptr->num = TmpState;

        memcpy(TmpState, SaveState, sizeof(STATETYPE)*num_states);

        for(idx=1; idx<=20; idx++) {
                if( (IsBad( LINK[idx] )) || (IsDormant( LINK[idx]) )) continue;
```

```
            /* save the link state */
            SaveLink = LINK[idx];

            LINK[idx] = 0;

            /* look and see if things are fully connected
               even following this link's failure
            */
            if(ConnectionFailure()) TmpUnsafeSum += LF;
            else {
              TmpCondSum += LF;
            }
            LINK[idx] = SaveLink;
        }
        atnodeptr->num = SaveState;

        if(IsUnderRepair(SUB[1])) TmpRecovSum += RR;
        if(IsUnderRepair(SUB[2])) TmpRecovSum += RR;

        /* report measurements */
        *SafeSum   = TmpSafeSum;
        *CondSum   = TmpCondSum;
        *UnsafeSum = TmpUnsafeSum;
        *RecovSum = TmpRecovSum;
}
```

```
SubCubeEffect(subid)
        int subid;
{
int idx,cnt,spare,base;

        /* enumerate number of working nodes */
        for(idx=1,cnt=0,base=(subid-1)*4; idx<=4; idx++)
                if( IsOK( NODE[base+idx] )) cnt++;
        if(cnt==4) return;

        /* at least one node is dead. Check the spare */
        if(subid==1) spare = 9;
        else spare = 10;

        if( IsDormant( NODE[spare] )) cnt++;
        if(cnt==4)  {
                /* the spare is OK. Mark the cube as under
                   recovery  */

                SetUnderRepair(SUB[subid]);
                return;
        }

        /* subcube failed */
        SetBad( SUB[subid] );
}

NodeEffect(nodeid)
 int nodeid;
{
int sb;
CONNPTR ptr1, ptr2;

        /* something in the node has failed. Figure out whether
           the node is still operative.
        */
        if( (IsGood( MEM_BUS[nodeid] )) &&
                (PORT[nodeid] > 2) && (PROC[nodeid] > 3)) return;

        /* this node is down. Drop all the components and tell the
            subcube manager
        */
        SetBad( NODE[nodeid]);
        SetBad( MEM_BUS[nodeid] );
        PROC[nodeid] = 0;
        PORT[nodeid] = 0;

        ptr1 = ptrarray[nodeid-1];
        while(ptr1) {
                SetBad( LINK[ptr1->link+1] );
                ptr1 = ptr1->next;
        }

        if(nodeid<9) sb = 1+(nodeid-1)/4;
        if(nodeid==9) sb  = 1;
        if(nodeid==10) sb = 2;
        SubCubeEffect(sb);
}

ProcEffect(nodeid)
 int nodeid;
{
        PROC[nodeid] = PROC[nodeid]-1;
```

```
        NodeEffect(nodeid);
}


Mem_BusEffect(nodeid)
 int nodeid;
{
        SetBad(MEM_BUS[nodeid]);
        NodeEffect(nodeid);
}


LinkEffect(linkid)
 int linkid;
{
int e1,e2;

        /* drop the state */
        SetBad(LINK[linkid]);

        /* decrement PORT count at endpoints */
        e1 = 1+LinkEnds[linkid-1].end1;
        e2 = 1+LinkEnds[linkid-1].end2;
        if( PORT[ e1 ]) PORT[ e1 ] = PORT[ e1 ] - 1;
        if( PORT[ e2 ]) PORT[ e2 ] = PORT[ e2 ] - 1;
        NodeEffect(e1);
        NodeEffect(e2);
}

SubCubeRepair(subid)
 int subid;
{
CONNPTR ptr;
int spare;
int cnt,idx,base;

        /* see if bringing up the spare will save the subcube */
        for(cnt=0,idx=1, base=(subid-1)*4; idx<=4; idx++)
          if( IsGood( NODE[base+idx] )) cnt++;

        /* this subcube is dead if either there are too few
           original nodes, or if the spare has already been
           swapped in
        */
        if(cnt < 3 || !(IsDormant( NODE[8+subid]))) {
                SetBad( SUB[subid] );
                return;
        }

        /*--- subcube is saved ---*/
        SUB[subid] = 1;
        spare = ((subid==1)?9:10);
        SetInUse(NODE[spare]);

        ptr = ptrarray[spare-1];
        /* bring the the links connecting the spare */
        while(ptr) {
                SetInUse(LINK[ptr->link+1]);
                ptr = ptr->next;
        }
}
```