IN-61-CR
110090
P- 36

Survivable Algorithms and Redundancy Management in NASA's Distributed Computing Systems

FINAL TECHNICAL REPORT

NASA Grant NAG9-426

for the period of
May 1, 1990 - April 30, 1992

Dr. Miroslaw Malek
The University of Texas at Austin
Department of Electrical and Computer Engineering
Austin, TX 78712-1084

Contents

List of Figures

List of Tables

1 Introduction

The design of survivable algorithms requires a solid foundation for executing them. While hardware techniques for fault-tolerant computing are relatively well understood, fault-tolerant operating systems, as well as fault-tolerant applications (survivable algorithms), are, by contrast, little understood, and much more work in this field is required. In this report, we outline some of our work that contributes to the foundation of ultrareliable operating systems and fault-tolerant algorithm design.

Our philosophy is based on the fundamental concept of consensus. For a system to be fault tolerant, there must be a multiplicity of resources and agreement among these resources on system status, be it concerning time or faults. In the next section, we outline our consensus-based framework for fault-tolerant system design. We believe that it is possible to develop a provably correct operating system nucleus, on top of which application-specific fault tolerance techniques are used. The development of the consensus-based framework and application-specific techniques for fault-tolerance are the core achievements of this project. These, of course, are in addition to our previous accomplishments in the formalization of fault tolerance, redundancy management, and hybrid algorithm methods for high performance and dependability.

In the next section, we introduce our consensus-based framework for fault-tolerant system design. This is followed by a description of a hierarchical partitioning method for efficient consensus. Section 4 introduces a scheduler for redundancy management, and application-specific fault tolerance is described in Section 5. In Section 6, we give an overview of our hybrid algorithm technique, which is an alternative to the formal approach given in Section 5. The report ends with Section 7, which is the summary and conclusions.

2 The Consensus–Based Framework

The consensus-based framework for fault-tolerant systems delineates the foundation and defines the principles for the specification, modeling, and design of fault-tolerant computer systems. We have defined the core, the nucleus concepts, and the functions that leads to comprehensive design methods for fault-tolerant computer systems.

Any successful design requires quantitative and/or qualitative goals that can be verified through measurement. The most successful designs are based on particular models that are accurate abstractions of reality. Of course, the ultimate model is a copy of the given system itself; however, with the high complexity of today's systems, such a model is frequently unattainable. Therefore, models for these systems tend to focus on a specific aspect of system behavior or a specific layer of system design. We concentrated on fault-tolerance and developed a layered model in which characteristics such as synchronicity, message order or lack of it, and bounded or unbounded communication delay are well defined for a specific environment. This layered model [14] is based on the consensus problem [2] and is, in our opinion, fundamental to the design of fault-tolerant multicomputer systems. In this case, consensus is defined as an agreement among computers. In multicomputer systems, the consensus problem is omnipresent. It is necessary for handling synchronization and reliable communication, and it appears in resource allocation, task scheduling, fault diagnosis, and reconfiguration. Consensus tasks take many forms in multicomputer systems.

Figure 1 is the model for fault management in a multicomputer environment in which each layer represents a separate consensus problem. At the base of the model is the synchronization level. For a system to be fault tolerant, there must be an agreement about time for fault detection and task execution. The next layer represents the requirement for reliable communication. Fault-tolerant computers must agree on how and when information is exchanged, and how many messages can be considered delivered or lost. The third layer, diagnosis, is fundamental to fault tolerance, for agreements must be reached on task scheduling and on who is faulty and who is not. Finally, the fourth layer illustrates the need for agreement on resource allocation and reconfiguration for efficient task execution and recovery from potential faults. In our fault-tolerant system design framework, we add an availability manager and application specific design methods that go on top of the kernel functions. This is shown in Figure 2. Another view of this framework is illustrated in Figure 3, in which functions in the kernel support applications armed with application specific techniques for fault-tolerant system design.

With the variety and complexity of the numerous applications in multi-computer systems today, we insist on this approach as it is our belief that general techniques have some limitations and, when used alone, cannot assure a high level of fault tolerance. We believe that, although the small generic kernel may be proved to be correct, the correctness of real-world applications, in most cases, cannot be proven. Hence, application specific techniques are necessary.

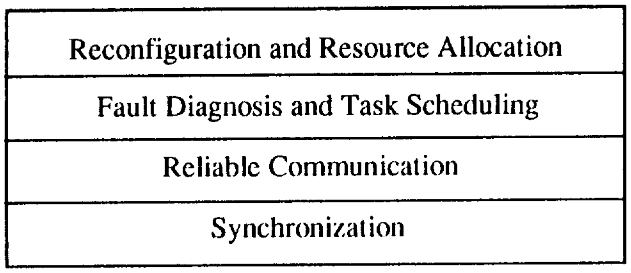

Reconfiguration and Resource Allocation
Fault Diagnosis and Task Scheduling
Reliable Communication
Synchronization

Figure 1: Consensus problems in fault management.

In fault-tolerant system design, all of the consensus problems should be accomplished in a timely and reliable manner. In order to design a fault-tolerant system, we need synchronization, communication, task scheduling, fault diagnosis, and reconfiguration. This means that each layer should incorporate algorithms to efficiently solve these tasks, as well as the techniques that cope with the various classes of faults.

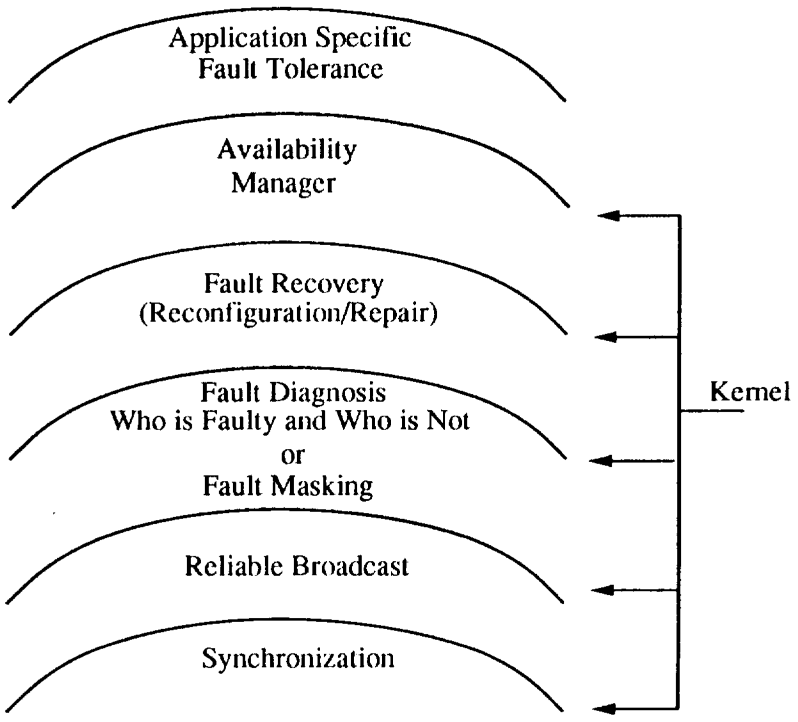


Figure 2: Framework for fault-tolerant systems design.

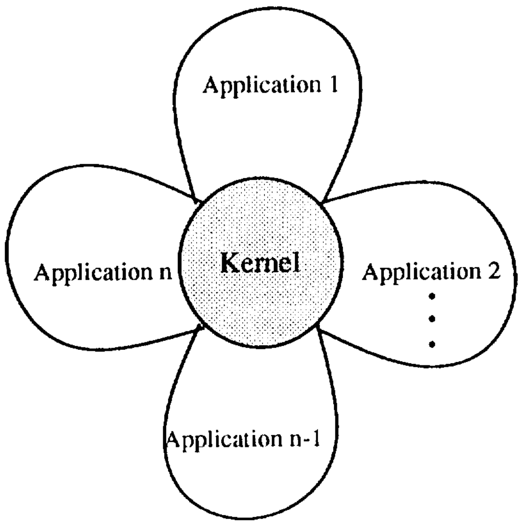


Figure 3: Another perspective on fault-tolerant systems design framework.

3 Efficient Consensus

In our design framework, we use consensus protocols to manage redundancy and to handle the diagnosis of and recovery from faults. Since any consensus protocol must operate in the presence of faults, that is, the protocol itself must be fault-tolerant, our primary concern is to make consensus protocols fault-tolerant and more efficient. This is achieved through the use of deterministic algorithms operating on a limited number of nodes, as is done in our Hierarchical Partitioning Method (HPM).

The HPM divides the system into many consensus partitions and organizes these into a hierarchy that permits efficient communication between the partitions. A partitioned system is a system that is divided into groups of k processors with each group running an internal consensus protocol. The HPM organizes the partitions hierarchically with separate consensus protocols for each group at each level of the hierarchy. For example, Figure 4 shows an $n = 27$ processor system divided into three levels of partitions, each containing $k = 3$ members. The final structure is a k-ary tree, k being the partition size, whose nodes are also partitions. The leaves of the tree, i.e., the lowest level, contain all the processors in the system in their partitions. At this lowest level, each processor is involved in its local consensus protocol. At the higher levels, only representatives from the lower levels are involved in the consensus. In this way, a global consensus is reached, although the information is distributed throughout the system. The hierarchical organization allows for the efficient retrieval of whatever part of this global information is required.

The driving assumption behind partitioning is that, in a large network, there will be groups of processors that, to a large extent, operate independently from other processors. In this case, global diagnosis and global consensus are not very useful. Therefore, we would like to create a mechanism that allows the formation of local consensuses, the reconfiguration of local consensuses, and the efficient dissemination of the results of other local consensuses. Hierarchical partitioning provides such a mechanism.

The HPM is a design for an implementation of consensus due to the choices a designer has in tailoring the HPM to a particular system. This flexibility includes choosing a particular consensus algorithm or set of algorithms that meet the needs of the system fault model (an extensive survey of consensus protocols may be found in [2]). For example, we have studied the HPM using system diagnosis techniques, which are consensus protocols designed to identify which processors are faulty and which are fault free,

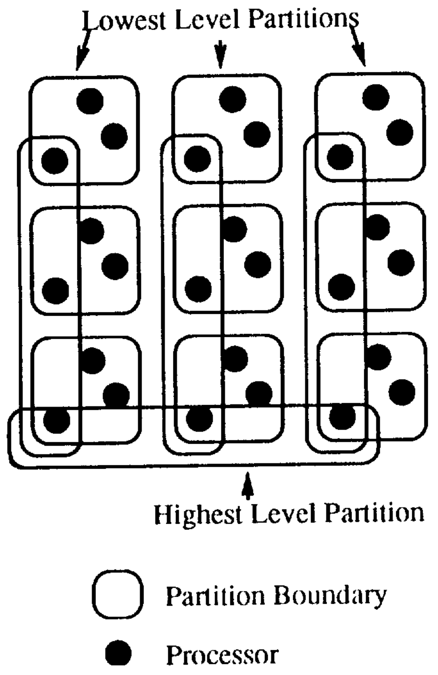


Figure 4: A 27-processor system partitioned into clusters of 3.

and Byzantine agreement algorithms, which are consensus protocols whose goal is to allow the fault-free processors to agree on some set of information [1]. We found that the HPM can greatly reduce the number of messages required to reach consensus. Figure 5 shows this savings [1]. As a result, there is a decrease in the time needed to reach consensus in the partitioned system over the time needed to reach consensus in the global, non-partitioned approach. This leaves more time for executing the system task set.

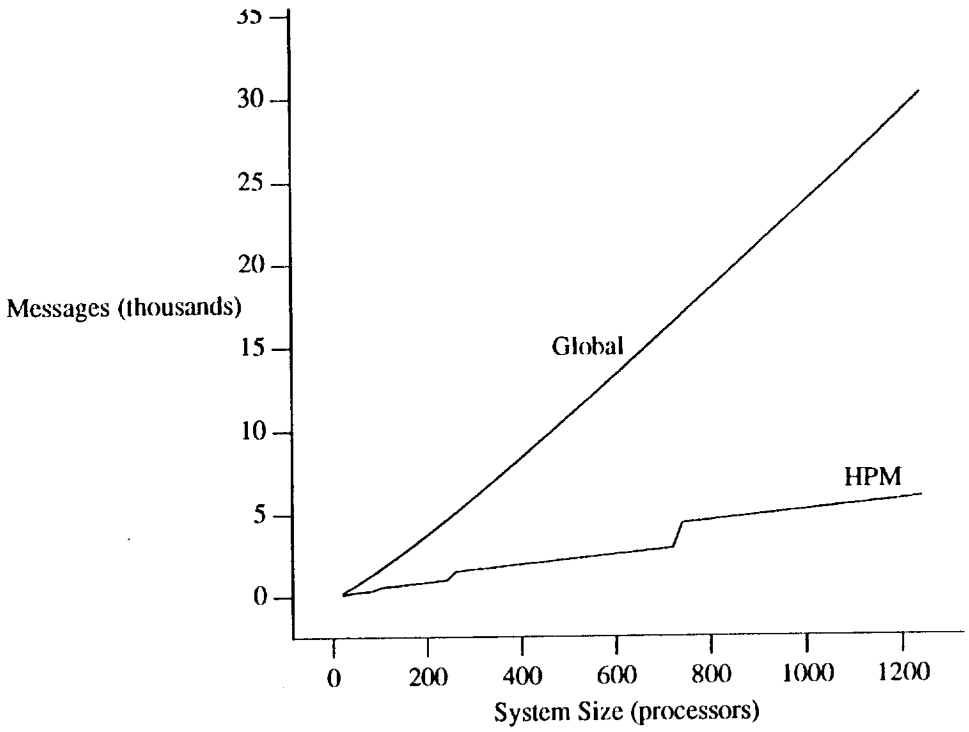


Figure 5: A graph of message count as a function of system size n for the HPM and the global consensus algorithm using system diagnosis techniques.

The drawback of decreasing message counts by partitioning is that maximum fault tolerance is decreased. That is, the maximum number of faults tolerable in a partition is related to the number of processors in the partition. Therefore, any partition containing less than the entire processor population limits the fault tolerance of the system. Yet, for large systems, it is not likely that the required availability restricts partitioning. In this case, we studied the effects of partitioning on the reliability of consensus, $R_{consensus}$, or the probability that correct consensus is reached in each and every partition [1].

We studied system diagnosis as it gives us more flexibility than Byzantine agreement, because the fault model allows the diagnosis and subsequent repair or removal of faulty processors. Once system diagnosis is complete, we can assume that the system is fault free. In terms of our measure $R_{consensus}$, $T_{consensus}$ represents the time between subsequent executions of the HPM using system diagnosis to achieve a certain reliability of consensus. That is, if the algorithm is scheduled every $T_{consensus}$ time units, then the rate of failure of processors should be such that, for each and every partition, the number of faulty processors is less than or equal to the maximum number of faults tolerable with probability $R_{consensus}$. The assumption here is that faulty processors are repaired at the end of each consensus period, thus, the system size remains constant.

We have examined how the Mean-Time-To-Failure (MTTF) of the processors, the number of processors n, and the size of partitions k affect the consensus period, $T_{consensus}$, required to meet a certain consensus reliability, $R_{consensus}$, for the HPM using system diagnosis. We assumed that each partition can diagnose at most t faults. Therefore, the reliability of a partition, $R_{partition}$, is the probability that no more than t processors will fail in that partition. That is,

$$R_{partition} = R_{PE}^{k} + \sum_{i=1}^{t} C_{k-i}^{k} R_{PE}^{k-i}(1 - R_{PE})^{i}; k > 1, t \leq k.$$

Given that the failure rate λ is the inverse of the MTTF,

$$R_{PE} = e^{-\lambda T_{consensus}}$$

in which $T_{consensus}$ is the consensus period. Also, given that $R_{consensus}$ is the probability that each of the n/k partitions is reliable, it follows that

$$R_{consensus} = R_{partition}^{n/k}$$

and, therefore,

$$R_{consensus} = \left[e^{-k\lambda T_{consensus}} + \sum_{i=1}^{t} C_{k-i}^{k} e^{-(k-i)\lambda T_{consensus}} (1 - e^{-\lambda T_{consensus}})^{i} \right]^{n/k} .$$

When t=1, this equation simplifies to

$$R_{consensus} = \left[(1-k)e^{-k\lambda T_{consensus}} + ke^{-(k-1)\lambda T_{consensus}}\right]^{n/k}.$$

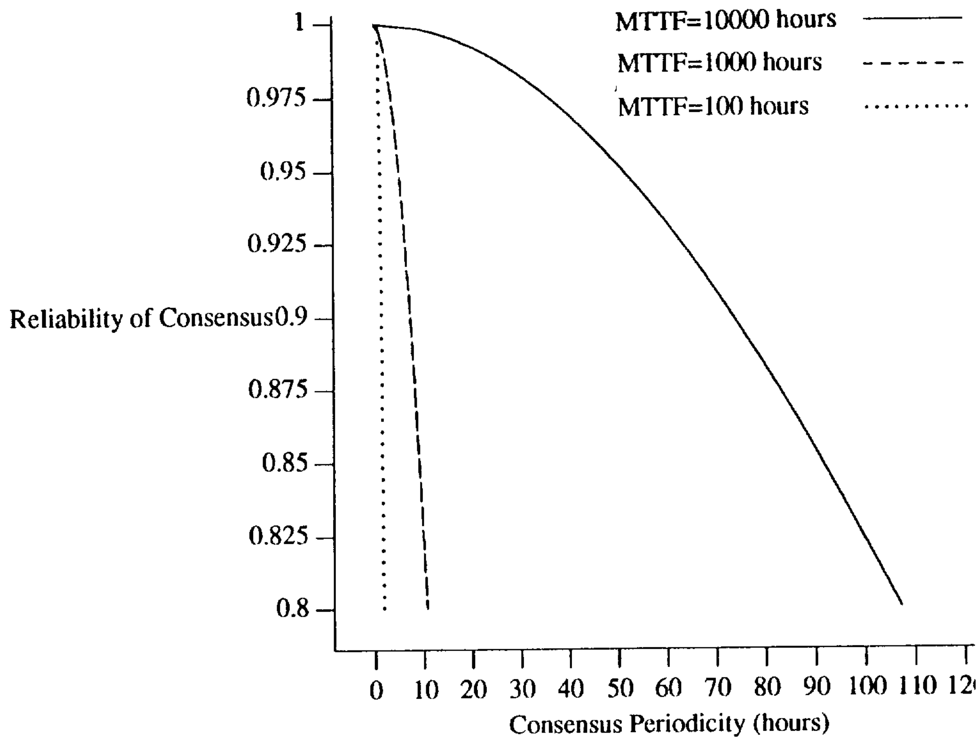
Using these last two equations, we can iteratively solve for the consensus periodicity $T_{consensus}$ for a given MTTF, system size, partition size, faults diagnosable per partition, and reliability of consensus. The following graphs show the effect on the consensus period of varying these values.

As an example, consider a 1000 processor system with partitions of size 5 that can each diagnose a single fault and whose processing and communication bandwidth allow the HPM using system diagnosis to be scheduled every 10 minutes (0.167 hours). The table in Figure 6 shows us that with processors whose MTTF is 100 hours we can expect a reliability of consensus between 0.99 and 0.999. If, on the other hand, the MTTF is 1000 hours then we can either increase the consensus period to between one and two hours or we can leave $T_{consensus}$ at 10 minutes and expect a consensus reliability better than 0.9999.

We introduced the Hierarchical Partitioning Method (HPM) to reduce the effects of reaching consensus in large, distributed systems, and we have shown that the HPM uses many fewer messages than a global consensus algorithm, which implies that it takes less time. Because consensus tasks are executed at the same time as other system tasks, they must not disrupt the network with large bursts of communication. The HPM divides the consensus into many independent tasks and keeps the consensus information distributed, thus avoiding the large message bursts that can occur in global consensus algorithms.

The HPM is a strong base on which to build highly fault-tolerant systems; it has an availability that is adjustable by the system designer, it reduces the time required to reach consensus by reducing the required number of messages and, thus, increasing the system's ability to produce timely results, and it may be based on any number of existing consensus protocols, which makes it flexible enough to suit the system's fault model. We are continuing to work towards a responsive (i.e., fault-tolerant and real-time) consensus algorithm based on the HPM that is improved in the areas of availability, timeliness, flexibility and efficiency, as well as in transparency, because a consensus mechanism should be available for any consensus task, including the consensus tasks of synchronization, communication, diagnosis and reconfiguration.

The importance of efficient consensus to our system design may be seen



Consensus Periodicity (hours)

	MTTF		
$R_{consensus}$	100	1000	10000
0.8	1.0731	10.7308	107.3079
0.85	0.9137	9.1367	91.3477
0.9	0.7337	7.3374	73.3736
0.95	0.5103	5.1028	51.0279
0.975	0.3577	3.5770	35.7694
0.99	0.2249	2.2493	22.4922
0.999	0.0720	0.7104	7.0850
0.9999	0.0224	0.2235	2.2412

Figure 6: Consensus periodicity as a function of required consensus reliability for various MTTF and $n = 1000, k = 5, t = 1$.

in Figure 2. The lowest four layers of our framework depend on the system's ability to reach a consensus among its processors. Therefore, the viability of our approach to fault-tolerant systems relies on our ability to produce an efficient consensus algorithm. We feel the HPM delivers an effective solution to this problem.

4 The Scheduler for Redundancy Management

The scheduler plays a critical role in the operation of a multiprocessor system, because scheduling in multiprocessor systems is the process of allocating resources to tasks so the tasks are executed efficiently. The reason that scheduling is widely studied is because, in general, it belongs to the class of NP-complete problems. Thus, a perfect solution to scheduling does not exist and scheduling policies or heuristics must be used. Since future systems will be complex and must operate correctly even in the presence of faults, the relative simplicity of the static scheduler must be relinquished, and, instead, dynamic scheduling, in which scheduling is performed "on-the-fly" as the tasks arrive, must be employed. A good design for the scheduler is essential, because it plays a central role in a fault-tolerant system. Not only is it relied on to arrange for the efficient execution of application tasks, but even fundamental system level tasks, such as executing programs to achieve synchronization or consensus on who is faulty and who is not, may have to be handled by the scheduler. It must also manage redundancy, allocate resources in the presence of faults, and be, itself, fault tolerant.

Scheduling for fault tolerance is a novel aspect that must be incorporated in highly fault-tolerant systems. The scheduler has to handle the issue of task fault tolerance. We expect the dependability requirement of all tasks to be specified. The system will attempt to achieve that requirement by adding redundancy to task execution when a processor cannot directly meet the specified goals. In our view, dependability can be achieved by close interaction between the scheduler and the Diagnosis and Recovery Layer (DRL). The DRL, at periodic intervals, updates the scheduler about the status of all processors (whether they are faulty or fault-free) and their dependability, such as their reliability or availability measure. This information is used by the scheduler to schedule the task to the appropriate location. However, if a critical task requires a dependability that cannot be met directly by a single processor, the scheduler attempts to form a processor group that meets this need through task execution redundancy that is based on the processor's dependabilities and fault models. There are two ways to add redundancy to a system, space redundancy and time redundancy.

Space redundancy is achieved by replicating the task over the processor group. Assume that a task that demands an availability $a_i(t)$ arrives, and its execution time has been estimated as τ and its time to deadline is d. In this case, the scheduler creates a processor group that has an availability of at least a_t over the time interval 0-d, and it schedules the replicas of the task on each of the processors of that group for τ time units in the interval 0–d. This information is then passed on to the DRL, which is responsible for forming a consensus about the result of the tasks and handling any faults in the replicas. Note that other tasks may also be scheduled on those processors over the remaining time.

Time redundancy is achieved by repeating the execution of a task on a single processor or by reconfiguring the processor group. Let us assume that a task has the same timing requirements as in the previous case. In the first case, the task may recover from a temporary fault if its execution is repeated. We can evaluate availability of such a task as $a_{12} = a_1a_2 + a_1(1 - a_2) + a_2(1 - a_1)$. Note that $a_1(t1)$ and $a_2(t2)$ may vary as they are executed at different times. In the second case, the DRL reports the availabilities of various processors. The scheduler selects the processor or processor group with availability of a_{pg}. This means that the processor or processor group is likely to be down for $1 - a_{pg}$ percent of the time d. Thus, the scheduler schedules the task for $\tau + (1 - a_{pg})d$ instead of τ time units, and, if a failure occurs and the processor group is down, there is still enough time for it to come up and recover from the fault and execute the task successfully.

A scheduler is itself a part of the fault-tolerant system and, as such, should be fault tolerant. Since the dynamic scheduling of tasks with non-deterministic characteristics on multi-processors is NP-complete, the time to obtain an optimum solution, if one exists, will be prohibitive. A scheduling policy or a scheduling heuristic would have to be used instead. However, one has to guarantee that the scheduler itself would obtain a schedule in a timely fashion. A scheduling policy such as First-come-first-served, Earliest-deadline-first, Least-laxity-first, etc., has the advantage of having deterministic times to schedule tasks, but a generic search-technique such as tabu [16], we believe, may be able to obtain acceptable schedules with a much lower development cost and a greater simplicity in design. In a complex system, several scheduling algorithms may need to be employed to achieve schedules of acceptable quality. It may also turn out that a generic search heuristic gives quality solutions while being simple and robust (in the sense of being able to solve any scheduling problem).

The scheduler, because it is at the core of an operational system, must be protected from faults. A scheduler failure is catastrophic, since no tasks can be executed while a scheduler is down, so it is necessary that the responsive scheduler be fault-tolerant. This means that there should be multiple locations where a scheduler is executing, so a single point failure cannot affect the entire system. An issue that has a direct bearing on this, as well as on performance, is whether the scheduler is centralized or distributed. A fault-tolerant centralized scheduler consists of multiple replicas, each of which cooperates to obtain a schedule. Each of these replicas can be identical, or each may execute a different scheduling algorithm, which would result in a hybrid search technique [15] [16]. For a distributed scheduler, each processor would have its own local or global scheduler that operates with the provision that the scheduler of another processor will take over in case of a failure. The local scheduler scheme requires a load-sharing strategy to handle additional load at a processor in case of transient overloads. Global schedulers need consensus to select the best schedule among the fault-free processors and, therefore, effectively manage redundancy.

4.1 Estimating the Number of Required Processors

An important issue that must be addressed when designing a system is to determine how many processors are required to meet system load requirements. We investigated the problem of determining probabilistically the the number of processors required in a real-time system based on the task characteristics — specifically, the interarrival time distribution, the execution time distribution, and the distribution of the time to deadline of the task. Assuming that none of these task characteristics are likely to be deterministic in a complex system, one would have to accept probabilistic estimates of how many processors are needed. In [18], we present a technique for obtaining such probabilistic estimates for an infinite-server queueing system that can provide an upper bound on the actual number of processors that may be needed.

4.2 Conclusions

The number of processors determined in the way described in Section 4.1 is an upper bound on the actual number of processors needed, which is largely dependent on the scheduling algorithm or policy used. For exponential interarrival time, we can exactly predict the number of processors needed for any distribution of execution times and the time to deadline. To determine the number of processors in such a case, one requires only the average execution time of the tasks and not the entire distribution of the execution times and the time to deadline. When the interarrival time is an arbitrary distribution that is not exponential, we have suggested an approximation to calculate the probability of the number of processors required. We have verified the correctness of our results by simulation of an infinite server queueing system. These results should be useful to designers of real-time systems in estimating the number of processors needed for an application. These results are also useful for predicting the number of processors for effective redundancy management under a variety of fault models.

5 Application-Specific Fault Tolerance

Fault tolerance usually requires redundancy in space (including hardware and software) or redundancy in time (see Fig. 7a). Our goal in this research was to achieve fault tolerance with low space/time overhead. In

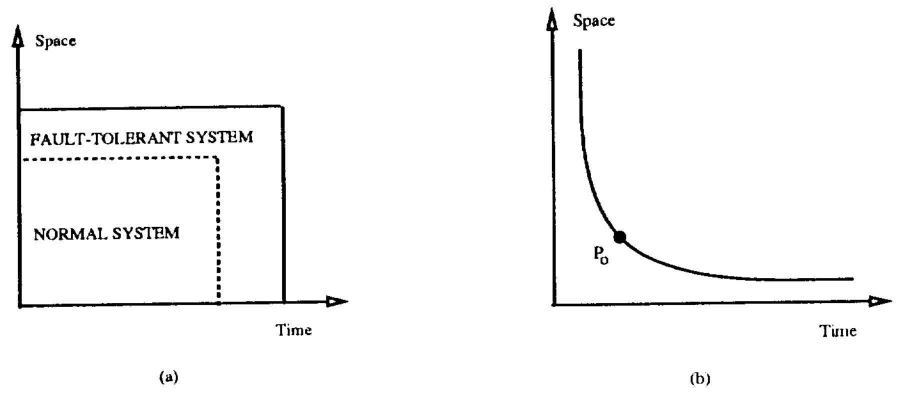


Figure 7: (a) Time and space overheads needed for fault-tolerant system implementation. (b) Desirable goal: fault tolerance with low space *and* time overheads.

our approach, we exploit application-specific properties that provide fault tolerance with low space and time overheads, in addition to classic, general methods in fault tolerance. We are not proposing that fault tolerance should be addressed *only* at the application level through the use of survivable algorithms. Rather, our thesis is that application-specific properties facilitating low-cost fault tolerance should *also* be considered in the design process along with other complementary techniques at the hardware or system level, such as self-checking or replicated logic, error detecting/correcting codes, checkpointing, and process/processor replication. We base our strategy for designing fault-tolerant applications on a comprehensive formalized scheme for fault tolerance called NEST [10].

The concepts investigated in NEST lead us propose a novel fault-tolerant technique based on the exploitation of *Natural Redundancy* in applications. It also facilitated the quantification of the space/time overheads incurred by existing fault-tolerant techniques.

5.1 NEST: A General Formalized Scheme for Fault Tolerance

The NEST scheme for fault-tolerant application design, described in [10], is based on a formal study of fault-tolerant algorithmic properties. These fault-tolerant properties may be provided at the hardware, system, or application level, but they are exploited at the application level. The formalization of fault-tolerant properties provides a common ground for studying fault-tolerant systems. In this context, redundancy is studied as a safety property and recovery is studied as a progress property. As a result, it is possible to define in a rigorous way what it means for an application to be fault tolerant.

Another consequence of this study is the outline of formal techniques to *add* fault-tolerant properties to applications when they are not present. This way, NEST provides both a model and a design methodology for fault-tolerant applications. Two algorithmic transformations, *superposition* and *concatenation*, are defined. Superposition can be used to add safety properties, such as redundancy, and concatenation can be used to to insert progress properties, such as recovery, into applications. The insertion of redundancy is called *invariant embedding* and the addition of recovery properties is called *progress securing*.

A complete description of NEST, including the formalization of fault-tolerant properties, a formal definition of application fault tolerance, and the proposition of a methodology for fault-tolerant parallel application design, is presented in [10].

5.2 Naturally Redundant Algorithms

It is obvious that the addition of redundancy and recovery procedures to an application will cause it to run with some time overhead. Since responsive systems must have fault tolerance and still meet deadlines, it would be nice to have applications or algorithms that are already redundant in some way. If these algorithms exist, one would need still to add a recovery procedure to them to make them fault tolerant, but no extra time overhead would be necessary to add redundancy. This idea lead to the following definition of a *Naturally Redundant Algorithm*:

Definition 5.1: If a given algorithm $\mathcal{A}$ maps an input vector $X = (x_1 x_2 ... x_n)$ to an output vector $Y = (y_1 y_2 ... y_m)$ and the *redundancy relation* $\{\forall y_i,\ y_i \in Y,\ \exists\ \mathcal{F}_i\ \mid\ y_i = \mathcal{F}_i(Y - \{y_i\})\}$ holds, than $\mathcal{A}$ is called a *Natu-*

rally Redundant Algorithm. Each $x_i(y_i)$ may be either a single component of the input (output) or a subvector of components.

From this definition, we can see that a naturally redundant algorithm running on a processor architecture P has at least the potential to restore the correct value of any single erroneous component y_i in its output vector. This will be the case when each $\mathcal{F}_i$ is a function of every $y_j, j \neq i$. If each $\mathcal{F}_i$ is a function of only a subset of the components of $Y - \{y_i\}$ then the algorithm would potentially be able to recover more than one erroneous y_i.

In many applications, processors communicate their intermediate calculations to other processors as the computation proceeds. In such cases, an erroneous intermediate calculation of a faulty processor, if allowed to be further disseminated throughout the architecture, can corrupt subsequent computations of other processors. It is thus desirable that the correct calculation value(s) be recovered before they are further propagated to other processors. This motivates the definition of algorithms that can be divided in phases that are themselves naturally redundant.

Definition 5.2: An algorithm $\mathcal{A}$ is called a *phase-wise naturally redundant algorithm* if (a) $\mathcal{A}$ can be divided in phases so the output vector of one phase is the input vector for the following phase, and (b) the output vector of each phase satisfies the redundancy relation.

We focused our attention on phase-wise naturally redundant algorithms. In order to use natural redundancy for achieving fault tolerance, we use mappings to a multiprocessor architecture so in each phase, the components of the phase output vector are computed independently (by different processors). Natural redundancy allows for a forward recovery approach, since there is no need to backtrack the computation to obtain the correct value for an erroneous output vector component. A naturally redundant algorithm can be made fault-tolerant by adding specific functionality to detect, locate, and recover from faults using its natural redundancy. In [11], two examples of naturally redundant algorithms, the solution of Laplace equations and the computation of the invariant distribution of Markov chains, are studied in depth. The results of the implementations are presented and discussed. The major advantage of exploiting natural redundancy is the ability to achieve fault tolerance with low performance degradation and small space/time overhead.

5.3 A Comprehensive Methodology for Fault-Tolerant Parallel Application Design

Based on the rigorous framework proposed in NEST, a comprehensive design methodology for fault-tolerant parallel applications can be summarized in three steps:

1. Clearly state system fault tolerance requirements and limitations.
2. Verify if the application (or an existing version of the algorithm) has inherent characteristics that cause some (or all) of the desired fault-tolerant properties to be met. If such properties exist, check if the fault tolerance thus provided meets the requirements of the previous step. If properties exist meeting all requirements then stop, otherwise execute the next procedure.
3. Apply general techniques that transform the existing version of the application so it acquires the missing properties and meets the desired fault tolerance related requirements.

In the first step, the designer should verify requirements such as (a) what classes of faults must be tolerated by the system, and (b) what are the acceptable cost levels, in terms of space and time overheads, the system can bear in order to achieve fault tolerance.

In the second step, the designer checks if the application (or an already existing version of the algorithm) is inherently fault tolerant, self stabilizing, has some natural redundancy, or any other characteristic that could facilitate a fault-tolerant design. If this is the case, it is still necessary to ensure that the fault tolerance resulting from these properties meets all systems requirements. For instance, if the intrinsic characteristics of the algorithm enables it to tolerate fail-stop faults, but multiple temporary faults are expected to affect the system, another fault-tolerant technique that can handle temporary faults must be used, and, if the intrinsic characteristics of the algorithm enable it to tolerate the classes of faults stated in the requirements but with higher time overhead than the system can bear, a more time-efficient fault-tolerant technique must be utilized. In summary, if some or all of the desired properties are missing or existing properties do not meet system requirements, the designer should apply general fault-tolerant techniques.

Step 3 aims to apply systematic transformation methods to an application or algorithm in order to add the missing fault-tolerant properties that will meet the desired requirements. These systematic transformations can be accomplished by the algorithm composition techniques studied in [10]. In order to insert redundancy, one would use the *invariant embedding* technique, which can be implemented by *algorithm superposition*. The practical issue here is to provide an invariant embedding that is both feasible and efficient to compute. Again, the specific characteristics of the application may favor one approach over several others. In order to add recovery procedures, one would use the technique we called *progress securing*, which can be implemented by *algorithm concatenation*. It should be noticed here that the type of redundancy (inherent or inserted to an algorithm) will largely determine the recovery procedures that may be implemented.

5.4 The Evaluation of Fault-Tolerant Techniques: The Cost/Benefit Relation

We evaluated a number of existing fault-tolerant techniques [12] for the space and time overheads they cause, and listed the kinds of faults they are able to tolerate. First, we discuss our model of computation. The techniques we cover are replication and voting [19], checkpointing and rollback [9], algorithm-based fault tolerance [7], self stabilization [5], inherent fault tolerance [3], and the approach based on natural redundancy [11].

In NEST, we adopted a model of computation that is based on the bulk-synchronous model of parallel computation proposed by Valiant [20]. In that model, the execution of a parallel algorithm proceeds in *supersteps*. The processes participating in a superstep are initially given a *step* of L time units to execute a specified amount of processing. After each period of L time units, a global check is performed to determine if the superstep has been completed by all participating processes. If that is the case, the computation advances to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. The model assumes the existence of facilities for a barrier synchronization of processes at regular intervals of L time units where L is the *periodicity parameter*. The value of L may be controlled by the program, even at runtime. This synchronization mechanism captures in a simple way the idea of global synchronization at a controllable level of coarseness. The realization of such a mechanism in hardware would provide an efficient way of implementing tightly synchronized parallel algorithms without overburdening the programmer.

In Table 1, the usefulness, in terms of tolerated faults, and the cost, in terms of space and time redundancy, for various fault-tolerant techniques is shown. In that table, N is the number of processors in the normal (non-fault-tolerant) version of the algorithm, and T is the total number of supersteps necessary for the execution of the normal algorithm in the absence of faults. Space redundancy is measured in terms of extra processors.

Replication with voting requires the largest amount of space overhead. Processors are at least triplicated. On the other hand, the time overhead is minimal. If a fault occurs in one superstep, recovery is executed in the next superstep. This technique covers a large set of faults, both temporary and permanent.

A considerable amount of space redundancy is also involved in the checkpointing and rollback technique. For each variable in the normal algorithm, some disk space must be allocated in the fault-tolerant execution to store the latest correct value for that variable. Evidently, extra code is necessary to do that, but no extra processes (or processors) are needed. The time redundancy required for recovery may vary depending on how far away, in terms of number of supersteps, the superstep in which the fault occurred is from the one in which the latest correct state was saved. An upper bound for this distance is I_{CP}, which is the interval, in terms of number of supersteps, between two checkpoints. This technique is usually used to tolerate temporary faults.

Algorithm-based fault tolerance, which has been mainly used with matrix problems, is accomplished with small space overhead and minimal time overhead. Two extra processors may be required to detect, locate, and correct single temporary faults, but basically only one extra superstep is necessary for recovery.

Self stabilization requires no space redundancy. After the occurrence of a fault, the computation can proceed from the resulting state and still reach the expected final results. On the other hand, the time redundancy necessary for the algorithm to converge after the occurrence of a fault is not predictable and may be quite large. In an experiment carried out in [11] with an iterative algorithm for solving Laplace equations, the time overhead varied between one extra iteration and 5.5 times the number of iterations necessary for the complete execution of the algorithm in the absence of faults. This overhead depends on how far, in terms of the number of iterations, the state resulting from the fault is from the fixed point. In [4], an experiment was done with a distributed system that was a restricted case of the problem proposed by Dijkstra in [5]. In that experiment, the number of state transitions and extra messages needed for the system to reach a correct state after a fault occurrence were $O(N^{1.5})$ and $O(N^2)$, respectively, in which N is the number of processes. Self-stabilizing algorithms can only tolerate temporary faults.

Inherent fault tolerance also requires no space redundancy. This type of fault-tolerant approach can only tolerate fail-stop faults. The occurrence of a fault causes a process to be permanently down (the processor stops). Since processors independently cooperate to achieve a common goal, and supposing that each processor contributes equally in this task, if one process fails, the upper bound on the number of extra supersteps necessary for the remaining processes to complete the job is equal to $\frac{T}{N-1}$. This upper bound is obtained calculating the number of supersteps necessary for $N-1$ processes to execute the complete algorithm (considering that N processes do it within T supersteps) and subtracting it from T.

In terms of extra work for the programmer, replication with voting, checkpointing and rollback, and algorithm-based fault tolerance require the algorithm to be redesigned to become fault tolerant. The main advantage of the self-stabilizing and the inherent fault tolerance approaches is that they impose no extra burden on the programmer. The approach based on natural redundancy falls somewhere between these extremes. It requires some extra coding to add a recovery procedure to the algorithm, but does not require the creation of redundant states.

For a naturally redundant algorithm to be made fault tolerant, there is no need for state extension or extra processes/processors (A characteristic of the algorithm is that its variables are already redundant algorithms in [11]). This technique requires no extra variables, processes, or processors, and has very low time overhead. Recovery is executed in one superstep that occurs immediately after the execution of the superstep affected by a fault. The fault coverage offered by this technique is also attractive. A naturally redundant algorithm can recover from both temporary and permanent single faults.

In terms of applicability, replication with voting, and checkpointing and rollback are generally applicable techniques. Algorithm-based fault tolerance, self stabilization, inherent fault tolerance and the approach based on natural redundancy are application specific.

One can intuitively perceive that there is a fundamental tradeoff in the design of fault-tolerant algorithms between space and time redundancy. For a given fault-tolerant technique, a higher space redundancy implies a lower time redundancy to tolerate faults. The converse is also true (see Figure 7b). This intuition is confirmed in practice when the diverse fault-tolerant techniques are compared. The replication with voting technique, which implies the largest space redundancy, requires minimum time overhead for recovery.

TYPE OF TECHNIQUE	REDUNDANCY				FAULTS TOLERATED
	SPACE # of processors		TIME # of supersteps		
	needed	extra	needed	extra	
Triplication with Voting	$3N$	$2N$	$T+1$	1	multiple temporary and permanent
Checkpointing and Rollback	N	—	$T+I_{CP}$	I_{CP}	multiple temporary
Algorithm-based Fault Tolerance	$N+2$	2	$T+1$	1	single temporary
Self Stabilization	N	—	?	?	multiple temporary
Inherent Fault Tolerance	N	—	$\frac{T*N}{N-1}$	$\frac{T}{N-1}$	multiple fail-stop
Approach Based on Natural Redundancy	N	—	$T+1$	1	single temporary and permanent

Table 1: Necessary space and time redundancy and faults tolerated by different fault-tolerant techniques.

On the other hand, the self-stabilizing technique, which requires virtually no space overhead, may incur a severe time redundancy. A balanced situation, corresponding to a fault-tolerant algorithm incurring low space *and* time overheads, could be represented by the point P_o in Figure 7b.

Considering the tradeoffs between the various fault-tolerant techniques, the approach based on natural redundancy, when this property is already present in the application, results in the most attractive cost/benefit ratio, if only single faults are likely to occur (which is true in most situations). It requires no state extension, only one superstep of time overhead, and provides high fault coverage at the cost of a small degree of algorithm redesign. The results listed in [11] fully support this claim.

5.5 Conclusions

The NEST predicate-based approach was introduced. It is a formal method of making algorithms fault tolerant. The NEST scheme was implemented, and a comparative analysis of a variety of fault-tolerance techniques was performed. Our technique, called naturally redundant algorithms, requires small time overhead, and can successfully tolerate single temporary and permanent faults. This approach is an attractive alternative to the Hybrid Algorithm Technique, which is introduced in the next section.

6 The Hybrid Algorithm Technique

The idea of combining two or more different algorithms into a single hybrid algorithm was inspired by the possibility that the new algorithm will perform better than any one of its component algorithms. The result is a new class of algorithms grouped under the umbrella of the hybrid algorithms technique(HAT). The hybrid algorithm technique combines the strengths of the individual algorithms so that the resulting algorithm has a combination of the following advantages:

1. it can produce better solutions,
2. it can produce solutions in less time,
3. it can tolerate software faults, and/or
4. it can effectively handle problems with larger input sizes, especially with respect to NP problems.

These advantages seem to be gained without major new disadvantages.

Figure 8 shows the basic idea underlying the HAT. Various algorithms cooperate towards performing a computation. At regular intervals, the results of the computation performed so far are compared by all algorithms and a good solution is distributed to all. This provides a very good mechanism for tolerating software or hardware faults, because any incorrect result will be weeded out during the consensus and exchange phase.

To demonstrate the capability of HAT, we have implemented a hybrid algorithm search technique for solving combinatorial optimization problems. To guarantee the optimum solution for these problems, all possible solutions must be considered. Unfortunately, many of these problems fall into the class of NP-complete, and therefore the set of all possible solutions is too large to consider. Heuristics are therefore used to test only the more promising subsets of the possible solutions. The existing algorithms cannot, therefore, assure that the optimum solution will be found.

Several algorithms exist that solve combinatorial optimization problems. Hybridization of some of these algorithms should combine the strengths of each algorithm's respective heuristic techniques and form a better algorithm, which ought to produce solutions that are closer to optimal, or in less time, or both. An algorithm that produces satisfactory results in less time can also be applied to larger problems.

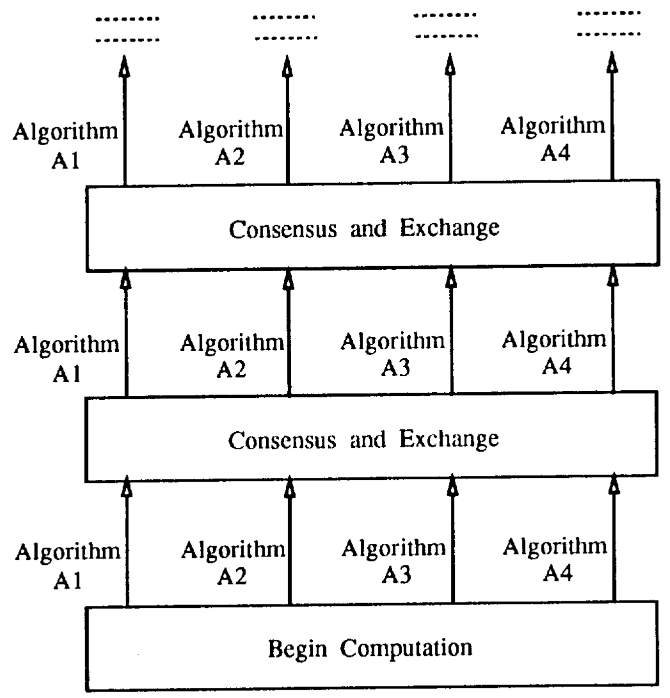


Figure 8: Overview of Hybrid Algorithm Technique

We expect our new hybrid algorithm search technique to be general and applicable to the majority of optimization problems. Some examples of problems where the hybrid algorithm search technique could be applied are in computer-aided design (e.g., integrated circuit or printed circuit board placement and routing), scheduling, resource allocation, test generation, integer programming, and a number of graph heuristic algorithms such as coloring and partitioning. To demonstrate the viability of our hypothesis of increased performance, we chose the Traveling Salesman Problem (TSP), which is an easily defined problem in combinatorial optimization research. The problem consists of finding the shortest Hamiltonian circuit (a circuit that includes every node) in a complete graph. The nodes of the graph represent cities and the edges are weighted with the distance between each pair of cities.

Our objective was to implement two different combinatorial optimization algorithms such that they may execute in parallel and exchange data periodically. The goal was to study the time efficiency and cost of mixing the simulated annealing [8] and tabu search [6] algorithms into a new parallel hybrid search algorithm with the costs of executing these algorithms independently. These three search algorithms, simulated annealing, tabu, and hybrid, were tested on the move of the 2-opt heuristic, which is based on swapping pairs of edges [16]. Experiments have been conducted on seven well known problems from the literature, namely, the 33 city, 42 city, 50 city, 57 city, 75 city, 100 city, and 532 city problems. Unlike the other problems, the 50 city and the 75 city problems have no known optimal solution.

6.1 Simulated Annealing/Tabu Search Hybrid (SATH)

Simulated annealing and tabu search use very different approaches to search for optimal solutions to combinatorial optimization problems. Although both of these algorithms provide good results on some problems, neither can guarantee the optimal solution will be found in real time. This, of course, leaves room for improved algorithms. We have therefore developed a hybrid algorithm in an attempt to produce better performance.

SATH is a simulated annealing/tabu search hybrid algorithm, the first in a new class of easily parallelizable hybrid algorithms. SATH incorporates both simulated annealing and tabu search as low level algorithms with a high level algorithm to mix the results from each. The idea is to execute each low level algorithm for some specified amount of time, the results of which are evaluated by the high level algorithm. The low level routines are then restarted in a more promising area of the solution space. This process is repeated as many times as is necessary or desired.

The SATH algorithm can be realized with the simulated annealing and tabu search portions implemented as subroutines. These subroutines could be executed, one after the other, followed by analysis of the results by a higher level routine. However, one of the most important features of this hybrid algorithm is the ease with which it may be executed in parallel. Each low level algorithm can be executed in parallel with a supervising process to synchronize execution and analyze results. This opens up the possibility of executing several low level algorithms in parallel, any number of which may be instances of simulated annealing or tabu search with different operating parameters. Interprocess communication is minimal and only occurs between a low level algorithm and the single high level algorithm. Speedup can therefore be linear with the number of processors as long as the number of processors does not exceed the number of low level algorithms.

6.2 Implementation of SATH

We implemented our SATH algorithm by allocating a separate process for each part of the algorithm. The basic implementation includes one main process and two child processes. When the program is executed, a main process is generated which reads in the problem definition. The main process then creates a set of child processes, one of which is a simulated annealing process, the other of which is a tabu search process. After specified time intervals, the child processes are halted and the main process compares their results. It selects a *good* solution for the child processes to continue with. A *good* solution might be the one with the least cost. If the tour with the least cost had already been given to the child processes, passing the same tour again will result in cycling. To prevent this from happening, the tour with the next to least tour (if not previously encountered) is made the common starting point for the child processes.

Other criteria might also be applied for defining a *good* solution. In our implementation, all the processes merge at a common point in the solution space when the tour with the least cost is distributed to all of them and is used as a starting point for the next iteration. Several other approaches might be considered, one of them being pseudorandomization. In this case, each process starts off with a pseudorandom tour after the information has been exchanged. This can be achieved by maintaining a history of the search space visited be each process in the previous iterations. Thus the new starting tours after the information exchange will be composed from previous history stored in the long term memory and information about the covered search space.

Implemented in this fashion, the SATH algorithm can be executed on a single processor or on multiple processors with very little effort. The algorithm is also expandable by adding additional simulated annealing and tabu search processes executing with different search parameters. The algorithm can be expanded in this way until there is a process for every available processor.

In our SATH algorithm, each simulated annealing process executes with a different annealing schedule. The schedules are chosen as in the accelerated simulated annealing algorithm described in [16]. When the SATH algorithm had multiple tabu search processes, each process had a different tabu condition and a corresponding tabu list size to distribute the search in the solution space.

6.3 Experimental Results

Our experiments with the traveling salesman problem have illustrated the advantages of using a hybrid search technique based on mixing simulated annealing and tabu search algorithms. The hybrid algorithm performs very well for all of the investigated problems, namely 33, 42, 50, 57, 75, 100 and 532 city problems. It holds considerable potential for reducing execution time for solving NP-complete problems and at the same time improving the quality of the solution. For a detailed description see [16] and [17]. With the advent of parallel processing in the computing environment, it becomes especially attractive to exploit the inherent parallelism in the proposed algorithm. A major advantage of the proposed approach is the ability to tolerate software faults due to multiple algorithm implementations. In addition, hardware faults can be tolerated in the multiprocessor implementation of the HAT. Further study of HAT will concentrate on the possibility of using genetic search algorithms for the selection/consensus phase of the algorithm. We strongly believe that this approach will further enhance the fault tolerance and performance of the HAT method.

7 Summary and Conclusion

As computer systems proliferate and our dependence on them increases, **fault tolerance** is becoming one of the most sought after qualities in computer and communication systems. Our research focused on the foundation for such systems using consensus, scheduling, and application-specific techniques to ensure effective redundancy management and the formal construction of survivable algorithms.

In our framework, the concepts of consensus and scheduling are fundamental. We have developed an efficient consensus algorithm based on the Hierarchical Partitioning Method. We have also specified a scheduler capable of reconfiguration even in the presence of faults, and we devised methods of estimating the number of processors to handle all tasks efficiently even in the presence of faults.

We pursued application specific methods for survivable algorithm design, because we strongly believe that high fault-tolerance can only be achieved by combining an ultrareliable kernel with application specific techniques. We also developed an alternative method, the hybrid algorithm technique, for making algorithms survivable. Our current research has been directed towards introducing fault tolerance in real-time systems. These fault-tolerant real-time systems, called *responsive systems* [13], are required for very critical applications, such as NASA's future Space Station. Redundancy management to obtain fault tolerance in such system is a challenging task due to the additional constraints of real-time and criticality of application. Our approach favors a comprehensive design of such systems, including specification, modeling, and design for redundancy management and recoverability.

In the future, the universal consensus algorithms for synchronization, reliable communication, diagnosis, and reconfiguration will be developed, and a scheduler that works in a reliable and timely manner even in the presence of faults will be implemented.

We believe that our research will have an impact on the design of future fault-tolerant, parallel/distributed systems, which aim for high availability, low space/time overhead, and effective integration of general and application-specific techniques.

References

[1] M. Barborak and M. Malek. Partitioning for efficient consensus. Technical report, The Department of Electrical and Computer Engineering, The University of Texas at Austin, May 1992.

[2] M. Barborak, M. Malek, and A.T. Dahbura. Consensus problem in fault-tolerant computing. Technical Report TR-91-40, The Department of Computer Sciences, The University of Texas at Austin, November 1991.

[3] F. B. Bastani, I. Yen, and I. Chen. A class of inherently fault-tolerant distributed systems. *IEEE Transactions on Software Engineering*, 14(10):1432–1442, October 1988.

[4] E. J. Chang, G. H. Gonnet, and D. Rotem. On the costs of self-stabilization. *Information Processing Letters*, 24:311–316, 1987.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[6] F. Glover. Tabu search methods in artificial intelligence and operations research. *ORSA Artificial Intelligence Newsletter*, 1, 1987.

[7] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Software Engineering*, SE-33(6):518–528, June 1984.

[8] S. Kirkpatrick, C. Gelatt, and M. Vechi. Optimization by simulated annealing. *Science*, 220(4598), May 13 1983.

[9] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[10] L. Laranjeira, M. Malek., and R Jenevein. NEST: A nested predicate scheme for fault tolerance. To appear in IEEE Transactions on Computers.

[11] L. Laranjeira, M. Malek, and R. Jenevein. On tolerating faults in naturally redundant algorithms. In *the 10th Symposium on Reliable Distributed Systems*, pages 118–127, September 1991. Pisa, Italy.

[12] L. Laranjeira, M. Malek, and R. Jenevein. Space/time overhead analysis and experiments with fault-tolerant techniques. In *The Third IFIP Working Conference on Dependable Computing for Critical Applications*, September 1992. Palermo, Italy.

[13] M. Malek. Responsive systems: A challenge for the nineties. In *Euromicro 90, Sixteenth Symposium on Microprocessing and Microprogramming*, pages 9–16, August 1990. Amsterdam, The Netherlands, Keynote Address.

[14] M. Malek. Responsive systems: A marriage between real time and fault tolerance. In *The 5th International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems*, September 1991. Nürnberg, Keynote Address.

[15] M. Malek, M. Guruswamy, H. Owens, and M. Pandya. A hybrid algorithm technique. Technical Report TR-89-06, The Department of Computer Sciences, The University of Texas at Austin, 1989.

[16] M. Malek, M. Guruswamy, H. Ownes, and M Pandya. Serial and parallel search techniques for the traveling salesman problem. *Annals of Operations Research*, 21:59 – 84, 1989.

[17] A. Mourad and M. Malek. Fault-tolerant parallel algorithm design. Technical report, The Department of Electrical and Computer Engineering, The University of Texas at Austin, August 1989.

[18] M. Pandya and M. Malek. Probabilistic estimation of the number of processors for real-time systems. Technical report, The Department of Electrical and Computer Engineering, The University of Texas at Austin, March 1992.

[19] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

[20] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.