Designing Fault Tolerant Applications in \textit{Maruti}

Deron Liang, Ashok Agrawala, Danniel Mosse and Yiheng Shi

System Design and Analysis Group
Computer Science Department
University of Maryland at College Park
College Park, Maryland 20742
{deron,agrawala,mosse}@cs.umd.edu

Abstract

In this paper, we introduce a model for developing applications with fault-tolerance requirements and real-time constraints. Applications in this model are specified using computation graphs, in which vertices represent tasks and arcs represent precedence constraints. Tasks are replicated to provide required fault tolerance and ensure that a real-time application will meet its deadlines despite failures.

We develop an analytical model to calculate the probability of successful execution of applications with task replications. We propose an efficient algorithm for the analysis of applications that are composed of subgraphs, each of which has a single source and a single sink. The results of the analysis can be used with information from allocation/scheduling to develop applications with desired timing and fault-tolerance requirements.

1 Introduction

Hard real-time systems are those in which not meeting an application deadline may lead to catastrophic consequences \cite{7}. Such systems are often used to support critical applications such as control of nuclear power plants, aircraft control, and space shuttle missions. Due to the criticality of these applications, fault tolerance techniques must be employed to ensure that its timing constraints are met despite failures.

Current real-time systems address fault tolerance in a rigid and static way, by fixing the number of redundant hardware components and statically allocating applications to the replicated hardware \cite{3,9,16}. Moreover, the actual systems incur extremely high overheads that make these systems inefficient \cite{15}.

In contrast, we specify a flexible mechanism for the users to adapt the design of the applications to their fault tolerance requirements. The user determines the number of replicas (i.e., redundant copies or software alternatives) for the applications, and the system builds a resilient application. However, since it is hard to determine the number of replicas for an application to meet the required fault tolerance levels, we need techniques that will aid the user in determining the number of replicas needed. We have also developed an analysis technique that estimates the reliability of an application, that is, the probability that the application will complete successfully within its deadline, with respect to the fault model. If a user application needs a reliability, the user should be able to gauge how well the application fits her/his goals. If the results are not satisfactory, the user should be able to tune the parameters to make the application resilient to the desired level. The tuning may consist of re-evaluating the reliability requirements, modifying the design of the application, or changing the underlying hardware platform (e.g., adding more resources).

A novel point about this work is the coupling of the resource allocation/scheduling with the fault tolerance scheme. This will enhance the resource utilization by allowing the system to take into consideration the fault tolerance requirements, in contrast with other systems that have a static allocation/scheduling scheme. The higher utilization comes from not allocating more resources than necessary for an application as in the fixed redundancy systems. Further, the linking of analysis and allocation/scheduling allows the system to decide on a mapping of the application to the resources of the system that maximizes the reliability.
of an application. Therefore, our approach addresses the problem of real-time fault tolerance in a single cohesive and comprehensive framework.

This paper is organized as follows: we first review the related works in the area of development of real-time fault-tolerant applications. In Section 2, we present the computation model of applications as well as the assumptions. In Section 3 we introduce an analysis technique for studying the reliability of resilient computation graphs and an example. Finally, we conclude this paper with the research contributions and future research.

1.1 Related Work

The most common approaches to fault-tolerance in distributed systems fall under one of two major categories: the primary/standby approach and the modular redundancy approach. In the first, recovery takes place after an error is detected, by switching from the primary to the standby. Depending on the real-time requirements of the applications, the latency\(^1\) for recovery may be too large to be admissible. Locus[9] and Auragen[3] are examples of the primary/standby approach. The primary/backup approach for software fault tolerance can be mapped to the recovery blocks approach [10], where a primary block of software is executed and if an error is detected, a recovery block (standby process) is invoked.

The second approach, namely modular redundancy, is better suited for real-time systems, due to its low latency for detection and recovery. In this approach a few modules execute identical functions in parallel and one result is selected using a message selection mechanism such as voting. If modules fail independently, a significant degree of fault tolerance is achieved by this approach. Since its onset, several hardware-based implementations of modular redundancy have been extensively examined in the literature [1, 16]. The main problems are the overhead and the lack of flexibility in specification of fault tolerance requirements, which may be excessive for some systems/users. For design faults in software, n-version programming (NVP) [2] is the main approach. In NVP, different versions of the software modules should be designed, implemented, and compiled using different sets of specifications, tools, and compilers. The results of different versions are compared at run-time, and a result selected. This introduces significant overhead, but decreases the latency for generation of correct results, when errors occur.

A number of efforts have been aimed at meeting the different requirements of fault tolerant hard real time systems, including SIFT[17], FTMP [13], FTP [14], and MAFT [6]. FTMP, MAFT and FTP require special hardware to perform specific fault tolerance tasks. SIFT, FTMP, and FTP were designed for specific applications, namely flight control, and provide little flexibility in their approach. The overhead for SIFT and FTMP is significant, utilizing 80% and 60%, respectively, of the system throughput to perform the fault tolerance functions.

It has been shown that the problem of estimating the reliability of resilient applications is equivalent to the problem of estimating the network reliability[8]. There is no known efficient algorithm to calculate the reliability of general networks. In [11], the model used to calculate the performance of fault-tolerant programs allows only sequential portions of program while executing on fail-stop processors [12]. In this paper, we also propose an efficient algorithm for the analysis of applications that can be recursively composed of subgraphs, each of which has single source and single sink\(^2\).

2 Models

In this section, before describing the computation and fault models, we present the framework on which our work is developed: the Maruti system. Maruti is an object-oriented system based on the analysis, allocation and scheduling of resources prior to run-time. For each task in an application, the start and end times are determined, and kept in a calendar. There is exactly one calendar per resource, created during the allocation/scheduling phase.

In the Maruti system, the user requests an application to be allocated and scheduled, triggering an allocation phase. The request for allocation includes the application itself, the timing constraints, and the fault tolerance requirements. The result of a successful allocation is the set of calendars that show that the application will be executed within its timing constraints. The dispatching of tasks is time-driven and based on the calendars, as opposed to event-driven

---

\(^1\)Latency is the time from detection of an error to the time when the system is again fully operational.

\(^2\)A fork-join task graph is one of such example.
systems. The current version of Maruti is a prototype implemented on top of UNIX, built for proof of concepts.

### 2.1 Computation Model

The user model of applications is a directed acyclic graph, computation graph (CG), in which vertices represent tasks and edges depict the relation between tasks. Tasks can be computation (\(T_i\)) or communication tasks between two computation tasks (\(C_j\), as in Figure 1. As part of the application characterization, we assume that task specification includes an upper-bounds description of the resource requirements\(^3\) [7]. The intertask relations can be interpreted as temporal relations, used for expressing the timing synchronization between tasks.

![Figure 1: (a) CG. (b) corresponding RCG](image)

In order to achieve the fault tolerance goals required by the application, each task \(T_i\) is replicated a number of times specified by the user, \(k_i\), and the interconnections among replicas established. The user may also specify interconnections among replicated tasks according to the criticality of the task and the resource availability. The resulting application is called resilient computation graph (RCG).

Replicas of the same task form a module \(M_i\), which corresponds to a task in the CG. The cardinality of \(M_i\) is denoted by \(k_i\). Therefore, the RCG is composed of replicas within a module (task \(T_{im}\) is the \(m^{th}\) task in module \(M_i\) and the communication among replicated tasks (\(C_{jm}\) represents the communication from task \(T_{im}\) to task \(T_{jm}\)).

\(^3\)Note that communication resources have to be treated analogously in our system.

### 2.2 Fault Model

Let us consider a distributed system which is composed of several multiprocessors connected via an underlying network. We assume that the resources in the system can be divided into non-overlapping groups of resources, called partitions, and that each partition is fault independent (i.e., the occurrence of an error will cause a failure in at most one partition). Each replica of a task \(T_{im}, m = 1, \ldots, k_i\) is allocated to a different partition, to enforce fault independence among replicas and to allow for load balance.

We assume fail-stop tasks, that is, a task produces correct output or none at all. For that, the system must provide a monitoring mechanism that detects the faults before generating results. Among the most common detection methods are alarms or watchdogs, signatures, and Acceptance Tests (AT)[4, 13].

The basic unit of failure in our model is the failure of a task. A task failure can be caused by failing hardware or software, exemplified by resource and design faults, respectively. Although our analysis is able to deal with several types of faults, for simplicity we
will consider here only software faults and transient hardware faults\(^4\). In our analysis, we assume that task failures are independent, that is, the occurrence of one failure does not affect the probability of occurrence of other failures.

We assume that the probability of individual task failure is known (i.e., we know the probability that a task will generate correct outputs given that the inputs are correct). These probabilities can be derived from software failure rates \([8]\) and transient failure rate of the hardware \([5]\), given that the start time and execution times for each task in the RCG is known (calendars). It should be noticed that since inter-task communications are also tasks in our computation model, their failures are also modeled as task failures.

2.3 Discussion

It should be noticed that fault tolerance is achieved at the cost of increased resource consumption. Therefore, a user must take into consideration the resource requirements of an application while specifying its resiliency degree \(k\). Furthermore, hard defined guarantees give little information about the relation between the probability of successful application completion and resource failure rates. It only guarantees that given up to a certain number of resource failures, it is still possible for an application to complete. Therefore, there is need to develop a method that analyzes the probability of successfully completing an application, given a temporal mapping from the tasks to the resources and the failure rates of individual resources. In the next section we describe such a method in detail.

The user may use the analytical results to iteratively decide on the replication level for each part of the CG. For example, a user may use the analytical results in estimating the number of replicas to be used in the construction of the RCG. The user can submit a resilient application to the system (say, with replication level \(\{k_i\}\)), and wait for the probabilistic analysis to generate a probability of success. Therefore, the number of replicas should be adjusted (increased or decreased) in order to fit the user’s needs. This procedure can be automated, and the user would, transparently, obtain a resilient application that satisfies his/her constraints.

---

\(^4\)Permanent hardware faults can be taken into consideration by a simple extension of our techniques.

3 Reliability Analysis of Resilient Graphs

In this section we present the reliability analysis of resilient computation graphs (RCG). At the end, we illustrate the application of this technique with an example. Before going into the details of the analysis, we first present the notations used in the discussion.

Notation

- \(M_i\): A module in the RCG that corresponds to a node in a CG. The cardinality of \(M_i\) is denoted by \(k_i\).
- \(T_{im}\): The \(m^{th}\) task (replica) in module \(M_i\);
- \(C_{ji}^{on}\): Communication from task \(T_{im}\) to task \(T_{ji}\);
- \(In(M_i)\): the Input-Set of module \(M_i\), that is, the set of modules that \(M_i\) receives input from;
- \(Out(M_i)\): the Output-Set of module \(M_i\), that is, the set of modules that \(M_i\) sends output to;
- \(R^T_{f}\): Success-Rate function of task \(T\) with respect to input-set \(I\), where \(T = T_{ij}\) or \(T = C_{ji}^{on}\);
- \(R^T_{M}\): the collection of success-rate function for module \(M\) with respect to input-set \(I\). That is, for each task \(T\) of \(M\), there is a success-rate function \(R^T_{f} \in R^T_{M}\);
- \(E(T)\): random boolean variable representing the correctness of the execution of task \(T\): \(E(T) = 1\ iff.\ \text{task } T\ \text{executes successfully;}
- \(O(T)\): random boolean variable representing the correctness of the output from task \(T\): \(O(T) = 1\ iff.\ \text{the output produced by task } T\ \text{is correct.}

To measure the probability of success of an RCG, we define a metric called (application) reliability. The reliability of a task is the probability that the output produced by this task is correct. Our goal is to obtain the reliabilities of all tasks at leaf-nodes (i.e., those that generate final results) of an RCG. Collectively, these reliabilities of tasks represent the probability that the application produces correct outputs. The reliabilities of RCGs are evaluated in terms of the conditional reliabilities of individual tasks on the condition that inputs to these tasks are correct.
3.1 Principles of the Analysis

For the analysis of applications depicted by resilient computation graphs, we take a decomposition approach. An RCG is first decomposed into single-entry and single-exit subgraphs (i.e., subgraphs that receive input from a single module, and send output to a single module). We analyze each of these subgraphs separately, deriving their (conditional) reliabilities on the condition that their inputs are correct. Each subgraph is then replaced by a pseudo-module $M_p$ with equivalent reliabilities. By recursively applying this process, the entire RCG is eventually reduced to a single pseudo-module with derived reliability. In general, any single-entry, single-exit subgraph can be replaced by a pseudo-module in this way. However, the reliability analysis of a single-entry, single-exit module can be very complex depending on its topological structure.

For the simplicity of the presentation, we focus our analysis on fork/join computation graphs, which can be generated by recursively applying one of the following two expansion rules: sequential, which expands a single module into two modules in tandem, and fork-join structure which expands a module into a fork-join structure, as shown in Figure 3. Note that by reversing the expansion rules, we can replace subgraphs with pseudo-modules, that preserve the characteristic of single-entry and single-exit modules.

It is important to notice that a task in an RCG can receive input from any replica in the invoking modules (input-set), each via a unique interconnection. A task can produce correct output only if it receives at least one correct input from each invoking module. However, due to the uniqueness of each communication task, exactly which tasks of each invoking module provides the correct input affects the reliability of the task invoked. Consequently, the conditional reliability of a task $T_{ji}$ depends on the status of its input-set, $In(M_j)$. In general, if a task $T_{ji}$ receives inputs from other tasks as shown in Figure 4, the conditional reliability of task $T_{ji}$ is a function of the status of those tasks from which it receives input, i.e.

$$R_{T_{ji}}^{In(M_j)} : dom(O(T_{j1})) \times \cdots \times dom(O(T_{nj})) \rightarrow [0,1]$$

where $dom(V)$ is the domain of variable $V$ (since $O(T_{ji})$ is a boolean variable, $dom(O(T_{ji})) = \{0,1\}$).

Now we are ready to analyze the (conditional) reliabilities for subgraphs. As we have seen, given a single-entry, single-exit subgraph, we can replace it with a pseudo-module $M_p$, without affecting the overall reliability of the application. Assume that the subgraph has entry module $M_f$ and exit module $M_f$ (exit and entry tasks are defined analogously). Then the pseudo-module $M_p$ is defined as follows.

- $In(M_p) = In(M_f)$;
- $Out(M_p) = Out(M_f)$;
- $M_p$ has the same number of tasks as $M_f$, that is, $|M_p| = |M_f|$; and
- $M_p$ has the same reliability functions as the subgraph, that is, $\forall T_{pi} \in M_p, R_{T_{pi}}^{In(M_p)} = R_{T_{pi}}^{In(M_f)}$.

Under certain circumstances, a subgraph with multiple entry modules and/or multiple exiting modules can also be replaced by a pseudo-module as defined above. For instance, if the input-sets of all entry modules are the same, they can be considered as a single entry module. Similarly, if all the exiting modules have the same output-set, they can be viewed as a single exiting module.

![Figure 3: Graph Expansion Rules](image_url)

![Figure 4: Communication Tasks as Part of the Receiving Task](image_url)
Figures 5 and 6 illustrate the transformation for sequential and fork-join subgraphs, respectively. We proceed with these transformations recursively, until we obtain a single pseudo-module, and thus we are able to compute the reliability of that pseudo-module. In the following section, we present the details of the analysis.

3.2 Analysis

In this section, we first derive the conditional reliability of a task \( T_{ji} \) as a function of its input set, \( \text{In}(M_j) \), as shown in Figure 4. Then, we demonstrate our merging technique to aggregate a sequential or a fork-join subgraph into a single pseudo-module.

Notice that task \( T_{ji} \) produces correct output only if the following conditions are satisfied (See Figure 4).

1. There exists at least one task, \( T_{im} \), for each module \( M_i \), for \( i = 1, \ldots, n \), whose output is correct and the execution of its corresponding communication task \( C_{ji}^{im} \) is also correct;

2. The execution of task \( T_{ji} \) is correct.

Based on these conditions and the independent failure assumption, we have

\[
\mathcal{O}(T_{ji}) = \left[ \prod_{i=1}^{n} \sum_{m=1}^{k_i} (\mathcal{O}(T_{im}) \mathcal{E}(C_{ji}^{im})) \right] \mathcal{E}(T_{ji}). \tag{1}
\]

Let \( r_{im} \) be a boolean variable that denotes that correct output was generated from task \( T_{im} \). For simplicity, we use \( P(b) = P(b = 1) \), when \( b \) is a boolean variable. From Equation (1)

\[
\begin{align*}
R_{T_{ji}}^{\text{In}(M_j)}(r_{11}, \ldots, r_{i_k}, \ldots, r_{n1}, \ldots, r_{nk_a}) \\
= P(\mathcal{O}(T_{ji}) = 1 | \mathcal{O}(T_{i1}) = r_{11}, \ldots, \mathcal{O}(T_{ik}) = r_{i_k}, \ldots, \mathcal{O}(T_{nk}) = r_{nk_a}) \\
= P \left( \left\{ \left[ r_{11} \land \mathcal{E}(C_{ji}^{11}) \right] \lor \ldots \lor \left[ r_{i_k} \land \mathcal{E}(C_{ji}^{i_k}) \right] \right\} \right) \\
\ldots \land \ldots \\
\left\{ \left[ r_{n1} \land \mathcal{E}(C_{ji}^{n1}) \right] \lor \ldots \lor \left[ r_{nk_a} \land \mathcal{E}(C_{ji}^{nk_a}) \right] \right\} \\
\land \mathcal{E}(T_{ji})
\end{align*}
\]

For simplicity, we also denote \( P(\mathcal{E}(T_{im})) \) as \( p_{im} \) and \( P(\mathcal{E}(C_{ji}^{im})) \) as \( q_{ji}^{im} \). Since the execution of tasks \( T_{im} \) and \( C_{ji}^{im} \) are independent by assumption, the above equation becomes

\[
R_{T_{ji}}^{\text{In}(M_j)}(r_{11}, \ldots, r_{nk_a}) \\
= \prod_{i=1}^{n} \left[ 1 - \prod_{m=1}^{k_i} (1 - q_{ji}^{im} \cdot r_{im}) \right] \cdot p_{ji} \tag{2}
\]

Since our goal is to replace subgraphs with pseudo-modules, we will first examine modules in tandem, and then analyze the fork-join structures.

(a) Two Modules in Tandem

We consider two modules executing in series as shown in Figure 5(a). We are interested in merging \( M_f \) and \( M_j \) into one pseudo-module \( M_p \), and derive the conditional reliability of each of its exit tasks, (i.e., \( R_{T_{ji}}^{\text{In}(M_j)} \), for \( l = 1, \ldots, k_j \)). First, we notice that the conditional reliability \( R_{T_{im}}^{\text{In}(M_j)} \), for \( m = 1, \ldots, k_f \), and
for \( 1 = 1, \ldots, k \), can be derived from Equation (2). Further, let \( \text{In}(M_f) = \{M_f\}_{i=1}^{n} \), where \( n \) is the cardinality of the input set \( (n = |\text{In}(M_f)|) \). We define events \( A, B \) and \( C \) such that:

\[
A = "O(T_{11}) = r_{11} \land \ldots \land O(T_{nk_k}) = r_{nk_k}" ,
\]
\[
B = "O(T_{1j}) = r_{1j} \land \ldots \land O(T_{j_{k_j}}) = r_{j_{k_j}}" ,
\]
and \( C = "O(T_{ji}) = 1" \).

Notice that \( R_{\text{In}(M_f)}^{\text{f}} = P(C|A) \). By law of total probability, we have

\[
P(C|A) = \sum_B [P(C|B|A) \cdot P(B|A)] \quad (3)
\]

We observe that conditional events \( (O(T_{jm}) = r_{jm} | A) \), for \( m = 1, \ldots, k_f \), are mutually independent due to the independent assumption on task failures. Hence,

\[
P(B|A) = \prod_{m=1}^{k_f} P(O(T_{jm}) = r_{jm} | A), \quad (4)
\]

where

\[
P(O(T_{jm}) = r_{jm} | A) = \begin{cases} 
R_{\text{In}(M_f)}^{\text{f}} & \text{if } r_{jm} = 1, \\
1 - R_{\text{In}(M_f)}^{\text{f}} & \text{if } r_{jm} = 0.
\end{cases}
\]

Further, we notice that conditional event \( (C|B) \) is independent of event \( A \), that is, \( P(C|B|A) = P(C|B) \) in Equation (3). From Equation (2), (3), and (4), we obtain Equation (5). (See next page.)

(b) Fork-Join Modules

We now show the merging technique for a fork-join module (Figure 6(a)), which takes place in two phases. In the first phase, we merge modules \( M_1, \ldots, M_n \) and \( M_f \) into a pseudo-module called \( M'_p \) (Figure 6(b)). Then, the fork module \( M_f \) and the newly created \( M'_p \) are executing in tandem. Therefore, they can be again merged into a single pseudo-module \( M_p \), as shown in Figure 6(b), using the same merging technique developed in the previous section (see Equation (5)).

To obtain the pseudo module \( M'_p \) in phase 1, we first apply Equation (2) to find the conditional reliability of tasks in the join module \( M_f \) \( (R_{\text{In}(M_f)}^{\text{f}}) \), for \( l = 1, \ldots, k_l \), as well as those of tasks in concurrent modules \( M_1, \ldots, M_n \) \( (R_{\text{In}(M_i)}^{\text{f}}) \), for \( i = 1, \ldots, n \) and \( m = 1, \ldots, k_i \). Since the concurrent modules share the same input set (i.e., \( \text{In}(M_1) = \ldots = \text{In}(M_n) \)), modules \( M_1, M_2, \ldots, M_n \) and \( M_f \) can be merged using a similar technique to that presented in the case of sequential modules. Namely, the conditional reliability, \( R_{\text{In}(M_i)}^{\text{f}} \), is given by Equation (6). (See next page.)

3.3 An Example

In this section, we analyze the reliability of a simple fault-tolerant application, with its CG and resiliency degrees \( \kappa_1 = \kappa_2 = 2 \), as shown in Figure 1(a). The application is transformed into the RCG as shown in Figure 1b by the fault-tolerance scheme described in earlier sections. The probabilities of successful execution of both computation tasks and inter-task communication are assumed to be 0.96, 0.98, respectively.

As previously described, we need to compute the reliabilities of the leaf nodes, which are \( T_{11} \) and \( T_{21} \) in our example. We present here the reliability analysis for task \( T_{11} \), since the analysis for \( T_2 \) is analogous.

Our first step is to calculate the conditional reliability function for each task. By applying Equation (2) of Section 3.2, we obtain the conditional reliabilities of the individual tasks, as the functions as listed in Table 7 & 8.

![Figure 7: Conditional Reliabilities of Tasks.](image)

<table>
<thead>
<tr>
<th>( T_{11} )</th>
<th>( R_{T_{11}}^{(M_f)} )</th>
<th>( R_{T_{12}}^{(M_f)} )</th>
<th>( R_{T_{21}}^{(M_f)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9408</td>
<td>0.9408</td>
<td>0.9408</td>
</tr>
<tr>
<td>0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

![Figure 8: Conditional Reliabilities of Individual Tasks](image)

Next we aggregate tasks \( T_{11}, T_{12}, T_{11}, T_{12}, T_{21} \), and their respective communication tasks (the lower half of the RCG) into a pseudo-module, as shown in Figure 6(b). By directly applying Equation (6), we obtain the conditional reliabilities of the pseudo-modules as shown in Table 9.
Figure 9: Conditional Reliabilities of the Pseudomodules

<table>
<thead>
<tr>
<th>$r_{fj}$</th>
<th>$R_T^{(M_f)}$</th>
<th>$R_T^{(M_T)}$</th>
<th>$R_T^{(M_T)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8800</td>
<td>0.8800</td>
<td>0.8339</td>
</tr>
<tr>
<td>0</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

After the aggregation, the original RCG becomes a graph with two modules in tandem. We then apply Equation (5), obtaining the reliabilities of tasks $T_{j1}$ and $T_{j2}$ as $R_T = R_{T_{j1}} = 0.8448$. Following the similar operations, we can obtain the reliability of the entire RCG, $R_M = 0.8966$.

In contrast, assuming the same task and communication failure rates, the application without replication has the reliability of $R = 0.7834$. If we change the parameters such that the second replicas $T_{j1}$ and $T_{j2}$ as well as the inter-task communication have lower failure rates, the results will reflect the modifications. For example, if $p_{T_{j1}} = 0.955$, $t_i = 1$, and $q_{T_{j1}} = 0.97$, $m 
eq i$, then the reliability of the CG and RCG are $0.7834$ and $0.8944$, respectively.

### 4 Conclusion

In this paper, we have presented the fault-tolerance scheme of Maruti, a distributed, fault-tolerant hard real-time operating system. Our design philosophy is to provide users with extensive flexibility in the development of fault-tolerant applications, while keeping the fault-tolerance scheme transparent to users.

Maruti also provides users with an evaluation tool to estimate the reliability of fault-tolerant applications. The evaluation takes into account of the computation graph of an application, its fault-tolerant requirements, and the resource allocation for each task. The analysis is based on graph decomposition. The tool is very useful in selecting efficient resiliency degree assignments to achieve desired reliability of the application with better resource utilization.

In our analysis, we have assumed that the failures of tasks in a module are independent. We notice that this assumption may not hold if the same copy of code is used for replication. We are currently investigating extensions to the model to cover cases with fault dependence. Secondly, we are also developing new techniques for a more general computation graphs.

### Appendix

#### A Algorithms

In this section we present the outline of the algorithm evaluating the reliability of an RCG. The algorithm consists of three sub-routines and a recursive graph-traversing main algorithm. The sub-routines compute the reliability function for a single module, a sequential subgraph, and a fork-join subgraph. These sub-routines are called $S.R.module$, $S.R.seq$ and $S.R.f-j$, respectively. The main routine traverses the graph to the deepest level, and recursively aggre-
gates the vertices into the pseudo-modules, computing their probability.

Algorithm 1 S.R.module\(M_0, \text{In}(M_0)\)

\[
\begin{align*}
\text{Begin} \\
(1) & \text{Compute the reliability function for each task:} \\
& \forall T_{0i} \in M_0 \text{ compute } R^{I_0}_{T_{0i}} \text{ (See Equation 2);} \\
(2) & \text{Return } R_{M_0}^{I_0} = \{R^{I_0}_{T_{0i}} \mid i = 1, \ldots, k_0\}. \\
\text{End}
\end{align*}
\]

Algorithm 2 S.R.seq\(\{R^{I_0}_{M_0}, R^{I_f}_{M_f}, M_f\}\)

\[
\begin{align*}
\text{Begin} \\
(1) & \text{Aggregate the subgraph into the pseudo-module } M_f: \\
& M_f = M_f \circ M_j \text{ (See Figure 5.)} \\
(2) & \text{Compute the reliability function for each task of } M_f: \\
& \forall T_{ji} \in M_j \text{ Compute } R^{I_{ji}}_{T_{ji}} \text{ (See Equation 5);} \\
(3) & \text{Return } R_{M_f}^{I_0} = \{R^{I_{ji}}_{T_{ji}} \mid i = 1, \ldots, k_f\}. \\
\text{End}
\end{align*}
\]

Algorithm 3 S.R.f-j\(\{R^{I_0}_{M_f}, \{R^{I_f}_{M_i} \mid i = 1, \ldots, m\}, R^{I_f}_{M_j}\}\)

\[
\begin{align*}
\text{Begin} \\
(1) & \text{Aggregate the subgraph into the pseudo-module } M_f: \\
& M_f = M_f \circ (M_1 \parallel \cdots \parallel M_m) \circ M_j \text{ (See Figure 6.)} \\
(2) & \text{Compute the reliability function for each task of } M_f: \\
& \forall T_{ji} \in M_j \text{ Compute } R^{I_{ji}}_{T_{ji}} \text{ (See Equation 2);} \\
(3) & \text{Return } R_{M_f}^{I_0} = \{R^{I_{ji}}_{T_{ji}} \mid i = 1, \ldots, k_f\}. \\
\text{End}
\end{align*}
\]

The main algorithm, Success, takes as input the entry module of an RCG and its input-set, and outputs the reliability function set of the RCG with respect to the input-set of the entry module. It proceeds by recursively aggregating subgraphs into pseudo-modules until the entire RCG is reduced into a single pseudo-module.

In the algorithm, we use the function \(Type(M_i)\) which identifies the type of module \(M_i\). In our analysis, a module could be of type SEQ (sequential), FORK, or JOIN. The function is formally defined as follows.

\[
Type(M_i) = \begin{cases} 
\text{SEQ} & \text{if } |\text{In}(M_i)| \leq 1 \text{ and } |\text{Out}(M_i)| \leq 1 \\
\text{FORK} & \text{if } |\text{In}(M_i)| \leq 1 \text{ and } |\text{Out}(M_i)| > 1 \\
\text{JOIN} & \text{if } |\text{In}(M_i)| > 1 
\end{cases}
\]

Algorithm 4 Success\(\{M_0, \text{In}(M_0), M^*_f\}\)

\[
\begin{align*}
\text{Begin} \\
[\text{Step1]} & \text{Analyze the Root Module: } S.R.module(M_0, \text{In}(M_0)) \\
[\text{Step2]} & \text{Process the First Subgraph: Aggregate It into Pseudo-Module } M^{(1)}_f: \\
& \text{Switch } (Type(M)) \\
& \quad \text{Case } \text{"SEQ"}: R_{M^{(1)}_f}^{I_0} = R_{M_0}^{I_0} \\
& \quad \text{Case } \text{"F-J"}, \text{"JOIN"}: \\
& \quad \quad \forall M_i \in \text{Out}(M_0) \\
& \quad \quad \text{Success}(M_i, \{M_0\}, M^{(1)}_f) \\
& \quad \quad \text{S.R.module}(M_j, \{M^{(1)}_f \mid i = 1, \ldots, m\}) \\
& \quad \quad \text{S.R.f-j}(R_{M^{(1)}_f}^{I_0}, \{R^{I_f}_{M_i} \mid i = 1, \ldots, m\}, R^{I_f}_{M_j}) \\
[\text{Step3]} & \text{Process the Second Subgraph: Aggregate It into Pseudo-Module } M^{(2)}_f: \\
& \quad \text{If } (\exists M'_0 \in \text{Out}(M^{(1)}_f) \text{ Type}(M'_0) \neq \text{"JOIN"}) \\
& \quad \quad \text{Success}(M'_0, \{M^{(1)}_f, M^{(2)}_f\}) \\
& \quad \quad \text{S.R.seq}(R_{M^{(1)}_f}^{I_0}, \{R_{M^{(2)}_f}^{I_0}, R_{M^{(2)}_f}^{I_f}, M_f\}) \\
& \quad \text{Else} \\
& \quad \quad M^*_f = M^{(2)}_f \\
[\text{Step4]} & \text{Final Result} \\
& \quad \text{Return } (R_{M^*_f}^{I_0}) \\
\text{End}
\end{align*}
\]

References


\(^\text{5}\) It can be shown that Out\(M^{(1)}_f\) contains at most one module.


