1

ADAPTABLE FAULT TOLERANCE FOR REAL-TIME SYSTEMS

A. Bondavalli, J. Stankovic*, L. Strigini**

CNUCE-CNR, Pisa, Italy

** University of Massachusetts, Amherst, USA*
*** IEI-CNR, Pisa, Italy*

ABSTRACT

This paper proposes a framework for software implemented, adaptive fault tolerance in a real-time context. It extends previous work in two main ways: by including features that explicitly address the real-time constraints; and by a flexible and adaptable control strategy for managing redundancy within application software modules. This redundancy management design is introduced as an intermediate level between the system design (which may itself consist of multiple levels of design) and the low-level, non-redundant application code. Application designers can specify fault tolerance strategies independently for the individual application modules, including adaptive strategies that take into account available resources, deadlines and observed faults. They can use appropriate design notations to notify the scheduling mechanisms regarding the relative importance of tasks, their timing requirements and both their worst case and actual usage of resources. Run-time efficiency can thus be improved while preserving a high degree of predictability of execution.

1 INTRODUCTION

This paper deals with the use of software implemented fault tolerance in those real-time, fault tolerant systems where a designer wishes to obtain more run-time flexibility than afforded by static redundancy and scheduling, while retaining the ability to depend on some guaranteed minimum level of performance. The reasons for this preference may include a highly variable application environment, the wish to provide an extensive and adaptable range of fault tolerance capabilities, and a high cost of computing resources. Examples of such

applications may be autonomous robots and weapon systems, or air traffic control. Software implemented fault tolerance typically involves substituting the execution of a software module with that of multiple modules, approximately implementing the same function, and executed on different processors and/or at different times. The management of error detection and treatment complicates both the tasks of writing the applications and of scheduling their executions. Therefore, systems with hard real-time requirements are often implemented with static redundancy (multiple copies of the software, producing a single result through either voting or self-checking and arbitration) and static scheduling. However, this approach is limiting for several reasons:

- in systems for complex, highly variable environments, static scheduling may not be the best way of using the limited resources available and may be too inflexible for adequate performance;
- this is the costliest form of fault tolerance in terms of resource consumption during fault free operation; adaptive schemes, with conditional execution of redundant modules upon detection of errors, are cheaper [3, 7, 12];
- especially if redundancy is meant to tolerate not just physical faults but design faults as well, the style of redundancy employed should be tailored to the individual application functions. So, the application designer should have some freedom in the composition of modules into redundant assemblies [19].

Dynamic scheduling has the general advantage of a more efficient multiplexing of the usage of resources by the various tasks in a system, exploiting the fact that the tasks in execution usually require less than their worst case resource allotment, and/or that it is admissible for some tasks to operate in a degraded mode in case of congestion. With adaptive fault tolerance, the difference between average and worst case resource requirements increases, increasing the advantages of dynamic scheduling. On the other hand, the execution time cost of dynamic scheduling is higher than for static scheduling so for tasks with very tight deadlines it may be necessary to use static scheduling. This paper proposes a framework for adaptive fault tolerance in a real-time context. It extends previous work in two main ways: one, by including features that explicitly address the real-time constraints; and two, by a flexible and adaptable control strategy for managing redundancy within application software modules. Application designers can independently specify fault tolerance strategies for the individual application modules, including adaptive strategies that take into account available resources, task importance, deadlines and observed faults.

They can use appropriate design notations to notify the scheduling mechanisms (both off-line and on-line) about the relative importance of tasks, their timing requirements and both their worst case and actual usage of resources. Run-time efficiency can thus be improved while preserving predictability[1]. Hence, this approach is suitable for highly dependable systems. In particular, the run-time support can reclaim those resources that the fault tolerance strategy, by necessity, leaves unused in all but the worst case fault conditions. The issues of minimum guaranteed performance and of robustness in extreme fault situations are addressed by the scheduling strategies using the notifications provided by the application designer.

In the rest of this paper, Section 2 describes our scenario of system development and execution. Section 3 details the syntax and semantics of our fault tolerant real-time structure (FERT). Section 4 deals with scheduling, and Section 5 contains our conclusions.

2 THE THREE-LEVEL FRAMEWORK

The design process of a real-time application typically starts with the specification of the physical inputs and outputs from/to the external world, a first specification of the important functional blocks in the system and the flow of data among them, and a first definition of timing requirements (periods and deadlines of tasks, response times of event-response chains). This top-level functional design ignores the issue of software redundancy. To manage redundancy, we introduce an intermediate level of design decomposition inside the functional blocks, and above the application code. Redundancy is added inside the individual functional blocks, using a general scheme called FERT (Fault tolerant Entity for Real Time). This effectively hides from the designer (both when doing the top-level algorithmic design and when designing the application code) the problem of managing redundancy. A FERT includes application modules (both the *functional* ones, which typically implement the black-box function of the FERT if the possibility of faults is not considered, and the adjudication modules, like voters) and a control part, which specifies the interactions of the application modules among themselves and with the scheduling support. The finished design includes three levels of functional decomposition (which must be consistent):

[1] Predictability is a complex term. See [18]. A simple definition usable in this paper is the ability to reliably determine whether an activity or set of activities will meet their requirements, in particular, deadlines.

- at the bottom, resides the code of the Application Modules (AMs). The AMs are the actual schedulable sequential tasks, seen by the run-time support; they execute in separate address spaces, both for protection and because they are often required to run on separate processors;
- at an intermediate level, we have the descriptions of the FERTs, each including a set of AMs and a Control Part;
- the top, is the system-level design, specifying:
  - the FERTs with their interactions: the indications of which message each FERT execution instance may send to each other FERT execution instance; these represent the *functional* design of the application and are used by the compiler to build the executable code and the appropriate dynamic binding information for message exchange; they also indicate some desired timing properties (e.g., a FERT producing a message should be scheduled before the FERT consuming it);
  - indication of which FERTs are *critical* in that their execution must be guaranteed off-line;
  - timing requirements: the periods of execution for periodic FERTs, the triggering events and minimum interval between executions for sporadic ones, and the deadlines of each execution;
  - a specification of mode changes, where each mode is described by a different combination of information as described above.

In proposing design notations, we do not mean that an entirely new programming language is needed. The concepts and notations we propose are somewhat independent of the programming and design description language used and we see no reason why they should not be implemented using an existing (programming or system design) language, with the application designer using either a subset of the language, or a macro language resembling our notation. In choosing such languages (and, of course, the in-the-small programming language could be used for system description as well) it is necessary to ensure consistency: for instance, mappings must be defined between the constructs through which the programmer of an AM specifies communication with the rest of the system, the constructs through which the FERT control notation specifies such interactions, and the inter-FERT communication primitives seen in the design description language. So, whenever possible we do not make any hypothesis about the language details (interprocess communication semantics, or structuring principles like objects, processes or data flow).

In the next section, to show in more detail how FERTs can be realised, we assume message-based communication, but it will be apparent that the FERT concept could be used with different languages and design styles.

Using the design and a description of the hardware, a static tool creates a reserved (but flexible) schedule that guarantees the minimum level of performance required under worst case assumptions on the AM execution times and the fault conditions (within the fault assumptions specified by the designer, e.g. maximum number of processor failures per time frame). If this step fails, the designer has to change the specifications or add hardware resources.

At run-time, a multi-level scheduling approach is used, that includes guaranteed on-time dispatching of reserved minimum performance, a planner that can arrange for execution of dynamic executions beyond the minimum reserved, and a resource reclaiming algorithm to reuse as much as possible of the time not used by reserved tasks (this *slack* time is produced either by (i) AMs finishing earlier than their worst case execution time, (ii) FERTs terminating with success without needing all the redundant executions planned for in the worst case fault hypotheses, or (iii) slack time already existing in the static minimum-performance schedule.

Each FERT typically allows more than one strategy, with different execution costs (e.g., triple vs. quadruple redundancy, or normal execution vs. execution of a minimal exception handler), to account for possible lack of resources at run-time. To decide between alternative possible schedules, the off-line and on-line algorithms attempt to maximise a utility function for the schedules they build. To this goal, the application designer has to specify an importance value for each FERT execution. We shall see later how this is inherited by the individual AM executions and how it can be modified under program control. It is thus possible for the application designer to specify general directives for scheduling, rather than individual decision rules for the huge number of decisions involved in scheduling.

The system (both off-line tools and run-time support) is then responsible for attempting optimisation. The word "attempting" is necessary because actual optimisation may be too complex even for an off-line algorithm, and the possibility of recomputing schedules at run-time is, of course, even more limited. However, for this design approach to make sense for dependable real-time computing, a requirement must be satisfied that the run-time scheduler behaves no worse than the off-line one. This requirement can be satisfied, for instance, by encoding the off-line minimum schedule into a table where a dynamic scheduler guarantees the minimum schedule and attempts to maximise use of the time

slots that are not allocated *a priori*. When considering real-time and dependability issues, it is impossible to avoid assumptions on the computing platform used. However, a practical design notation should not be restricted to any specific run-time platform (hardware and kernel). Our scenario assumes a run-time platform able to guarantee some basic run-time predictability and protection: it must be able to *stipulate* and *honor* contracts for timely execution. Global system-wide management and optimisation are performed obeying the hints given by the application designer, and limited to the capabilities of the individual platform. Within these constraints, different platforms can be used, leading to different levels of performance for a given application, but all guaranteeing predictable execution. The platform should also generally guarantee a clean failure semantics for the run-time support in the presence of hardware faults, including a global time base and interprocess communication, and watchdog timers guaranteeing that tasks do not exceed their allotted execution time.

3 FERT SPECIFICATION LANGUAGE

3.1 Control Plus Application Modules

The structure of a FERT is depicted in Figure 1. The Control component, interacting with the real-time kernel, specifies the interactions between the Application Modules (AMs) and between the FERT and its environment. Typically, the control part would describe a fault tolerance strategy employing the AMs, e.g., multiple version programming, recovery blocks, SCOP [3] or other more ad hoc designs needed for a specific FERT. This novel approach allows the designer to specify several alternative fault tolerance policies, taking into account the run-time state of the system and the availability of resources, including time. The FERT designer considers Application Modules as basic components with procedure-like interfaces. The FERT's interfaces with the rest of the system through unidirectional (input or output) ports. These may be of different types which account for the types of the data exchanged, size of associated queues, failure semantics and message ordering disciplines. Port types and routines for communication management are defined in libraries available to the FERT designer. Ports may also carry control information. Control output ports are used to signal self-estimates on the service provided by the FERT, or to propagate control information to other FERTs. Control input ports contain the actual control parameters which constrain individual executions of FERTs. They help the designer express decisions on how to use the resources available at any given instant. Two control input ports are defined for all FERTs,

carrying the deadline and importance of the required service; a designer may decide to have additional control information passed to the FERT through user specified input ports.

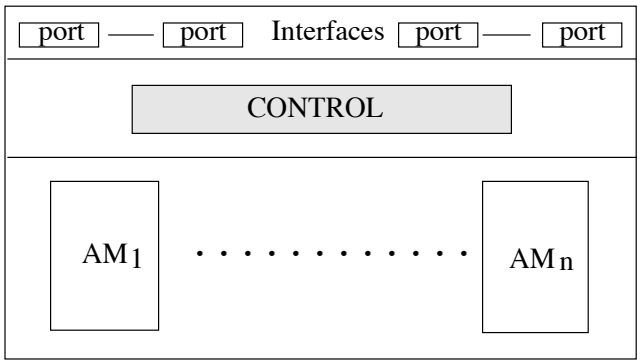

Figure 1 Structure of a FERT.

The input ports for the deadline and importance parameters have special semantics: if no control parameters are supplied through them, the deadline and importance associated to a FERT at execution time default to those specified in the design phase, and input primitives return these default values. So, control information is always available at control input interfaces, and the designer does not need to distinguish whether default values, or those provided by some other FERT, are used in a given execution. Constraints are also enforced on the values which can be sent through these control ports, to limit the run-time authority of FERTs on the control parameters of other FERTs, in accordance with global policies. The importance of a task for the system is expressed with two items of information, the gain (VALUE) obtained through a correct execution and the loss (PENALTY) incurred by missing this correct execution. Both VALUE and PENALTY appear at the FERT control interface, while, inside the FERT, the control uses just one value: positive, representing a gain, or negative, representing a loss for the system.

The designer of the FERT sees the **control** component as an algorithm, which manages the available redundancy by interacting with the underlying machine for real-time management. It specifies a set of possible alternative strategies. At run-time, it asks the real-time kernel which strategies can be supported. If the kernel issues a *guarantee* for a certain strategy, it means that it is able to schedule the AMs involved in that strategy, so that they finish before its deadline. Then the control chooses one among the supported strategies, and controls the execution of the AMs necessary for that strategy, taking appropri-

ate decisions on the basis of the observed errors. In practice we do not plan to have this algorithm compiled as executable application code; rather, it is compiled to a table used at run-time by the real-time kernel supporting the FERT's execution.

3.2 Control Language

Our FERT control notation uses four special primitives (POSSIBLE, EXEC, UNUSED, OUTPUT). The control algorithm has the structure of an Alternative Guarded command (in CSP style [6]). Each guarded command corresponds to one of several alternative strategies devised by the designer, and exactly one strategy must be selected and executed at FERT activation. Each strategy has its own control parameters (value and deadline), assigned by the designer to represent the service quality obtainable by using that strategy; if executed, all its AMs have to terminate execution by the same deadline, and the gain or loss for the system is defined for the entire strategy. The value and deadline for a strategy may be specified as functions of the FERT's value, penalty and deadline. Obvious constraints are: i) the deadline cannot be later than the deadline of the whole FERT; ii) the value cannot be greater than the VALUE associated to the whole FERT, nor lower than the PENALTY. A negative value means that the execution of that strategy represents a loss for the system. The values and deadlines of the strategies allow the real-time kernel to make tradeoffs between the request of different FERTs, trying to optimise the total value of a schedule.

Guards are based on a schedulability condition of the AMs necessary for the corresponding strategy, expressed by the primitive POSSIBLE:

```
[  POSSIBLE (s1p, ATs1, s1b, ATs1, s2, ATs2)
      with Highvalue by Deadline --> Strategy 1
[] POSSIBLE (s1p, ATs1, s1b, ATs1)
      with Mediumvalue by Deadline --> Strategy 2
[] POSSIBLE (s1p, ATs1, s2, ATs2)
      with Lowvalue by Deadline --> Strategy 3
[] POSSIBLE (...)
      with Lowestvalue by Deadline --> Strategy 4 ]
```

The parameters of the function POSSIBLE may specify precedence constraints: (AM1, AM2) specifies sequential ordering, while (AM1 || AM2) does not impose any order between AM1 and AM2. By calling POSSIBLE, the Control asks the planner part of the real-time kernel to accommodate the complete execution of the listed AMs before the specified deadline. Subject to the precedence constraints specified by the separators, plus those implied by the code of the AMs, the planner tries to build such a schedule, considering both the value associated to each strategy of the FERT and the fact that it must accommodate one of the strategies of the FERT. If this attempt is successful the function POSSIBLE returns true. The POSSIBLE function with the empty parameter list, in the last line of the example, always evaluates to TRUE: the corresponding strategy must be limited to sending a failure message, if the time necessary can be accounted for in the dispatching of the FERT, and it naturally has the lowest value among all the strategies.

In the specification of a strategy, the designer uses:

- **EXEC** (List of AMs (actual parameters)) to request the execution of the listed AMs, and provide the actual parameters for the execution of each AM. Each actual parameter may be:
  - a data item from an input port of the FERT
  - a reference to an input port
  - an output from the execution of another AM
  - a constant.

  The outputs of an AM can be used as input parameters for other AMs, or output of the whole FERT. **EXEC(s1p (Datain), ATs1 (s1p.O))**, e.g., first requests the execution of the AM **s1p** with the actual parameter taken from the input data port **Datain**, and then the execution of the acceptance test **ATs1** on **s1p.O**.
- **UNUSED** (List of AMs) to notify the resource reclaiming part of the real-time kernel that the control has decided not to execute some AMs whose execution had previously been planned. This notification should be performed as soon as the control decides not to execute some scheduled AM. Take, as an example, strategy 1 in Figure 3. If, after **EXEC(s1p (Datain), ATs1 (s1p.O))**, the acceptance test **ATs1** shows that the execution of the primary **s1p** has been correct, the control decides that the strategy has been successful and the backup **s1b** and the acceptance test **ATs1** do not need to be executed. This is specified by **UNUSED**

(s1b, ATs1). Depending on the kind of real-time kernel, this information can be useful or completely ignored. Still, our notation allows the designer to signal to the real-time kernel that it can reclaim resources (processors etc.), conservatively allocated to components that then end up not being executed, and give these resources to other components, thus improving the overall performance of the system. The resources seen by the control designer are simply the AMs of the FERT, rather than the actual run-time resources (processor, locks, files, etc.) needed by the AMs. The underlying kernel takes care of reclaiming these resources.

- **OUTPUT** (Value, Port) to commit the execution of a strategy, and hence of the FERT, writing the FERT results to the output ports. Values can be constants, decided by the control itself, or the results available from any AM executed. In **OUTPUT (<s1p.O, Result>,<Success, Control_O>)** the control is writing the output **O** of **s1p** to the FERT output port **Result** and the constant **Success** to the port **Control_O**.

3.3 Example FERT

We now show an example of a FERT meant to demonstrate (1) adaptability, (2) flexibility, (3) how the strategies are a function of real-time constraints, and (4) time-constrained run-time behaviour. In this example, the computation is divided into two parts, following the imprecise computation model. The first sub computation releases an approximate result and the second, starting from the approximate result, improves its precision. Moreover, to improve reliability, a primary and a backup variants, s1p and s1b, have been defined for the first stage, together with acceptance tests for the two stages, ATs1 and ATs2. The example is shown in Figures 2 and 3.

As seen in Fig. 3, the first strategy is the most complete one, exploiting all the redundancy available. If the system is not able to support it, the choice is between the second strategy, which is more conservative, producing only an approximate result, but with a higher probability than the third, and the third strategy which possibly produces a precise result, but may also lead more easily to complete failure. If the system cannot support any execution, the fourth strategy simply signals the failure of the FERT. Given the set of AMs in the FERT, many strategies can be designed. It is the designer's business to select meaningful ones according to the fault semantics he chooses for his FERTs.

```
FERT IC_RB
Data input: Datain: Type1;
Data output: Result: Type2;
Control input: Deadline: time, Value: integer, Penalty: integer;
Control output: Control_O: {Success, Failure, Exception1};

AM   s1p (input : I: Type1,
          output: O: Type2);
     MaxExT = 10, Resources=..., Prec =...;
     <s1p body>; (* the first stage primary *)

AM   s1b (input : I: Type1,
          output: O: Type2);
     MaxExT = 12, Resources=.., Prec =...;
     <s1b body>; (* the first stage backup *)

AM   ATs1 (input: Altres: Type2,
           output: Judgement: {Success, Failure});
     MaxExT = 3, Resources=..., Prec =...;
     <ATs1 body>; (* the first stage AT *)

AM   s2 (input : I: Type2,
         output: O: Type2);
     MaxExT = 10, Resources=..., Prec =...;
     <s2 body>; (* the second stage *)

AM   ATs2 (input: Altres: Type2,
           output: Judgement: {Success, Failure});
     MaxExT = 3, Resources=..., Prec =...;
     <ATs2 body>; (* the second stage AT *)
```

Figure 2. FERT header.

```
Control:
[  POSSIBLE (s1p, ATs1, s1b, ATs1, s2, ATs2)
        with Highvalue by Deadline --> Strategy1
[] POSSIBLE (s1p, ATs1, s1b, ATs1)
        with Mediumvalue by Deadline  --> Strategy2
[] POSSIBLE (s1p, ATs1, s2, ATs2)
        with Lowvalue by Deadline   --> Strategy3
[] POSSIBLE (...)
        with Lowestvalue by Deadline --> Strategy 4 ]

Strategy 1:
Begin
EXEC(s1p(Datain), ATs1(s1p.O));
If ATs1.Judgement = Success
        then Begin UNUSED (s1b, ATs1)
                EXEC(s2(s1p.O), ATs2(s2.O))
                If ATs2.Judgement = Success
                then OUTPUT (<s2.O, Result>,
                 <ATs2.Judgement, Control_O>);
                else OUTPUT (<s1p.O, Result>,
                 <Exception1, Control_O>);
              end
        else Begin EXEC(s1b(Datain), ATs1(s1b.O));
             If ATs1.Judgement = Success
             then Begin EXEC(s2(s1b.O),ATs2(s2.O))
                  If ATs2.Judgement = Success
                  then OUTPUT (<s2.O, Result>,
                     <ATs2.Judgement, Control_O>);
                  else  OUTPUT (<s1b.O, Result>,
                     <Exception1, Control_O>);
                  end
             else Begin OUTPUT (<Failure, Control_O>);
                        UNUSED ( s2,ATs2);
                  end
end
```

Figure 3a: FERT Specification of Strategies

```
Strategy 2:
Begin
EXEC(s1p(Datain), ATs1(s1p.O));
If ATs1.Judgement = Success
   then Begin UNUSED (s1b, ATs1)
        OUTPUT (<s1p.O, Result>, <Exception1,
                Control_O>);
        end
   else Begin EXEC(s1b(Datain), ATs1(s1b.O));
        If ATs1.Judgement = Success
        then  OUTPUT (<s1b.O, Result>,
              <Exception1, Control_O>);
        else OUTPUT (<Failure,Control_O>);
        end
end

Strategy 3:
Begin
EXEC(s1p(Datain), ATs1(s1p.O));
If ATs1.Judgement = Success
   then Begin
        EXEC(s2(s1p.O), ATs2(s2.O))
        If ATs2.Judgement = Success
        then OUTPUT (<s2.O, Result>,
             <ATs2.Judgement, Control_O>);
        else OUTPUT (<s1p.O,Result>,
             <Exception1,Control_O>);
        end
    else Begin OUTPUT (<Failure, Control_O>);
         UNUSED ( s2, ATs2);
         end
end

Strategy 4:
OUTPUT (<Failure, Control_O>);
```

Figure 3b. FERT: Specification of Strategies.

4 SUPPORT FOR SCHEDULING

It is necessary to integrate the off-line scheduling decisions with the dynamic operation of the system in a manner such that the off-line guarantees are not violated at run-time, and such that the system maximises its effectiveness beyond the minimum guaranteed part.

4.1 Off-line Guarantee Algorithm

Off-line support is necessary to create *a priori* guarantees that the minimum performance and reliability of the system are achieved. This may be accomplished in a number of ways. Here, we briefly discuss one way to accomplish this task using a form of reservation of flexible time slots. The off-line algorithm works with the following assumptions:

- a specification language describes the timing and fault behaviour of modules on an individual basis as well as other module requirements such as precedence constraints and general resource requirements;
- a system-level specification details the minimum level of guaranteed performance and reliability required;
- the workload requirements are specified, and
- some knowledge of the run-time algorithms and environment is utilised.

The environment information includes the hardware resources available and a distributed, real-time, fault tolerant system kernel that has (i) a global time base, (ii) run-time data structures that contain the flexibility and adaptability requirements of the application, (iii) predictable primitives, (iv) run-time scheduling support, and (v) a basic guarantee paradigm which uses on-line planning. The off-line guarantee algorithm takes as input all the information listed above and attempts to find a feasible allocation and schedule for the modules that are part of the minimum guaranteed requirements, as well as accounting for other requirements in order to obtain good performance beyond the minimum. In particular, the interaction between the FERT specifications and the off-line algorithm is as follows. FERTs are typed as being critical or non-critical. All critical FERTs are guaranteed by the off-line scheduling algorithm. In many cases the critical FERTs will have a single strategy defined and therefore this is what must be guaranteed. If more than one strategy is defined for a critical FERT, then the minimum strategy is *a priori* guaranteed by

the off-line algorithm, and at run-time, if it is possible, a more comprehensive and valuable strategy may be dynamically guaranteed each time the FERT executes. Non-critical FERTs are dynamically guaranteed using the options specified in the FERT, but some overall time and resource availability may be guaranteed for all non-critical FERTs. Since this algorithm executes off-line and for the critical tasks, significant compute time can and should be devoted to this problem. If the heuristic is having difficulty in producing feasible allocations and schedules, the designer can choose to add resources to the system or modify requirements and re-run the algorithm. The output is a flexible time table with earliest and latest scheduled start times, and finish times for all the critical tasks and their minimum redundant copies and/or voters, and idle intervals. This table is used in a flexible way by the 3 on-line algorithms described in the next subsection.

Various heuristics for static allocation and scheduling exist in the literature [8, 15, 21]. We base the discussion of what is required in a new heuristic on a set of extensions to the heuristic found in [15]. That heuristic is able to schedule complex periodic tasks in distributed systems. It handles periodic time constraints, worst case execution time, general resource needs, precedence constraints, communicating tasks, and replication requirements. The communicating tasks, when allocated across nodes of the distributed system, are scheduled in conjunction with a time-slotted subnet. The algorithm as it now exists has been implemented and evaluated by simulation. Further, extensions to the algorithm have been developed which attempt to balance load and spread out (in time) scheduled tasks to avoid clustered computation time which could cause long latency. In other words, some results have been developed which account for the dynamic operation of the system when performing the static allocation and scheduling off-line. For our purposes, the current algorithm must be enhanced in the following ways:

- considering aperiodic tasks with minimum guarantees,
- enhancing the fault semantics to include those supported by FERTs,
- accounting for the dynamics in a more sophisticated manner including creating a window for each statically guaranteed task composed of an earliest start time, latest start time and deadline,
- addressing tasks of different importance, and
- addressing mode changes.

All of these changes can be made to the current algorithm. The algorithm should be part of an interactive tool that aids the designer in the off-line design and analysis.

4.2 On-line Scheduling Support

The on-line scheduling support includes dispatching, resource reclaiming, and planning, all cooperating to provide adaptability, robustness, and predictability, where predictability includes meeting performance and fault handling requirements. Here we show that predictable dispatching and resource reclaiming have largely been solved, but that significant problems still exist with planning.

Dispatching

It is possible to develop off-line scheduling analysis and algorithms that produce a table where each guaranteed task (a task is a dispatchable entity and may be part of a FERT) has a window in which to execute (given by an earliest and latest start time). The dispatcher is a simple (execution time bounded) mechanism that knows how to deal with such a time table. The computed execution time for all tasks in the system includes dispatching and resource reclaiming costs, so when a task completes there is always time to see if resources can be reclaimed and to dispatch the next task. The dispatcher may wait (or idle) if no task is ready, e.g., because the next task is a periodic task whose arrival time has not yet been achieved. Dispatch lists are on a per processor basis so that there is no locking overhead if N processors attempt to dispatch simultaneously. In summary, what is required is a predictable dispatching for multiprocessors where resource reclaiming is possible. Such a dispatching mechanism exists in the Spring kernel [17] and only requires simple updates to handle flexible starting times.

Resource Reclaiming

To support both predictability and flexibility, on-line dynamic planning is used. When plans are dynamically created the schedules are constructed using worst case execution times and shared data and resource requirements. The purpose of resource reclaiming is to collect previously reserved time (in all resources) that is no longer needed for various reasons such as: a task has completed before its worst case execution time, redundant copies of a computation are no longer needed, a mode change deleted the need for certain tasks, etc. The time

collected appears as idle time in the table and can be used by the dispatcher in moving computations forward (some care is needed here, to avoid various anomalies [16]), or by the planner in either (1) scheduling newly arriving work, or (2) reinstating additional redundancy that possibly could not be utilised prior to this point in time. For a given system, not all these options need to be exercised, e.g., in many systems point (2) may not be practical for implementation. The basic resource reclaiming required for adaptive fault tolerance has already been developed, implemented, and evaluated [16], again in the Spring kernel. The implementation achieves good performance with low and bounded implementation costs with a key property being that anomalies are prevented when reclaiming resources (time).

Planning

The planning algorithm runs in parallel with application processes. It uses the *a priori* generated flexible time table (that accounts for all resources, not just the CPU) to insert newly invoked work (above the minimum reserved). If newly invoked work can be placed in the table, then the work is dynamically guaranteed, else various actions are taken based on the current policy. The planner also uses the on-line descriptions of the fault behaviour of the active FERTs, compiled from their control components. For example, suppose a certain FERT is invoked, and the planner identifies the preferred strategy as requiring a primary and 2 backups to be scheduled prior to the deadline, all on different nodes. If the planner can find open intervals for this requirement, in time, then that strategy is dynamically guaranteed. If not, the planner applies the designer specified action, e.g., the information associated with this FERT might indicate just to abort the FERT, or alternatively it might indicate that a strategy consisting of a simple error handler with no redundancy should be scheduled. The planner for a given system must be sufficiently powerful to support the level of adaptability of fault semantics specified by the designer, and, in general, planners on different nodes must cooperate to find feasible task assignments to time slots, including subnet time slots. A key problem is making the planner fast enough (and bounded) to be usable in many systems. For example, the cost of distributed planning may be reduced by using a scheduling chip [4] and replicated global memory based on ScramNet such as in the SpringNet system. These architectural features should expand the application domain of applicability of adaptive fault tolerance.

5 CONCLUSIONS AND DISCUSSION

Many real-time applications require a high degree of fault tolerance. Software implemented fault tolerance offers advantages in terms of flexibility of error treatment and effectiveness against design faults. However, it is surprising that very few papers explicitly address real-time scheduling to meet timing and fault tolerance requirements. These include [2, 9, 10]. These papers are valuable for highly static, embedded computer systems where fault tolerance is extremely important and deadlines are very tight. In such cases, guaranteeing primary and contingency schedules as in [9, 10] is paramount. Integrating solutions such as these into environments where more flexible responses might be required is a key research problem. For example, the static schedules found in these solutions can only handle preconceived types of failures and may catastrophically fail under some unexpected failure, overloads, or correlated failures. In dynamic environments, the solutions should if possible adapt to current system conditions and gracefully degrade under unexpected events. It is also important to note that these referenced papers assume that the entire task completes, in effect, producing a correct value at the end of the computation. This may often be too restrictive.

The topic of flexible and adaptive fault tolerance has been addressed by a few authors. The need to allow per module specification of software redundancy has been recognised often (for instance, in the Delta-4 system [14]: extensions to include design diversity are discussed in [5]). [19] argues the usefulness of flexible schemes of software fault tolerance, besides the straightforward N-version programming and recovery block schemes; instances of such schemes have begun to appear in the literature (e.g., [20]). A notation for structuring software with different forms of software fault tolerance is proposed in [12]: this is less general than ours in that the *control* part has to be chosen among a few permitted kinds. In [1], the application programmer is supposed to provide the diverse software modules used for software fault tolerance, and directives to static tools which create a recovery meta-program. The latter manages module execution, including state savings and rollback, through appropriate kernel calls. This scheme allows the programmer to employ software fault tolerance without strong restrictions on the structure of the application program. However, real-time issues are not considered.

Adaptive schemes that allow run-time decisions about the degree and form of redundancy are in [11, 12] and [3]. In the latter, the designer can control the execution time by setting the maximum allowable number of execution rounds, but no mechanism is provided for the translation between rounds of computa-

tion and execution time. No one has previously studied the combination of: i) different redundant schemes in different run-time modules; ii) adaptivity of redundancy management to the available resources, and iii) interaction between strategies local to each module and system-level optimisation, so as to allow a departure from completely static design, which may be too inflexible and too costly for demanding application environments.

This paper has presented a general framework and a specific notation, called FERT, for real-time, adaptive, software fault tolerance. Its novelty consists in addressing jointly the three requirements above, by including a flexible control strategy for error handling and explicit features addressing the real-time constraints. The designers of the FERTs have great freedom in specifying redundant execution strategies. By specifying alternative strategies, and using the POSSIBLE, EXEC, UNUSED, and OUTPUT primitives, they can interact with the on-line and off-line scheduling policies to select strategies, and control their execution, adapting to the actual load and fault situations. The scheduler may thus manage the available resources, seeking an optimal usage subject to the requirements of the application designers. The application designers can use the flexible notations for masking most errors inside the FERTs.

The requirements on the run-time support are meant to ensure a consistent, simple semantics of the notation used in designing a FERT. All types of hardware faults to be tolerated are visible at the FERT level as consistent (non-Byzantine) value, timing or omission errors of the AMs, as are software faults inside the AMs themselves. So, the designers can use the FERTs as fault containment units, masking hardware faults via the distribution of the AMs, and application software faults via a strategy appropriate for the semantics of the AMs. At a higher level in the design, FERTs can therefore be treated as being fault-free or having clean failure semantics. Last, fault treatment via reconfiguration (for long mission times with delayed repair) can be attained using mode changes.

In this paper, we have restricted ourselves to the non-recursive use of FERTs. Although technical problems have to be solved, we have shown the feasibility of the approach. Alternatively one could use an expanded, recursively composed FERT notation as the only design language for whole complex systems. This seems an attractive way of structuring large, complex systems allowing the designer at each level of decomposition to address both functional and redundancy management aspects of the design. However, such a fully recursive approach integrated with predictable and flexible real-time scheduling is far from being achievable at the current state of the art. Two major problems would have to be faced and solved. The first is the need for a dynamic planner with the

ability to perform planning with bounded delay and a short actual absolute run-time. The second problem is a matter of convenience. Since guarantees imply pessimism, even having such a bounded-delay planner, the schedules produced would often be useless. The pessimistic estimates would be propagated through the recursive composition tree and the schedule would include all the worst case resource requirements. Therefore, in many situations no schedule would be found, and in the others the utilisation of reserved resources would be unaffordably low. We have argued the practicality of these design concepts by showing that the scheduling mechanisms that are needed to exploit their potential are feasible through extensions to existing methods and algorithms. It is still necessary to fully develop the extensions and to demonstrate their cost effectiveness for real applications. To demonstrate effectiveness we need to consider many issues including coordinated scheduling across nodes in a distributed system, integration with proper hardware support, and a possible relaxation of the notion of guarantee (when acceptable) in order to improve overall performance. Developing good metrics that adequately represent the flexible and adaptable properties of the system is also required.

Acknowledgements

This work was supported in part by the Commission of the European Communities, in the framework of the ESPRIT Basic Research Project 6362 Predictably Dependable Computing Systems (PDCS2), and in part by the National Science Foundation under grants IRI 9208920 and CDA 8922572, and the Office of Naval Research under grant N00014-92-J-1048.

REFERENCES

[1] M. Ancona, A. Clematis, G. Dodero, E. B. Fernandez and V. Gianuzzi, "A System Architecture for Fault Tolerance in Concurrent Software," IEEE Computer, Vol. 23, pp. 23-31, 1990.

[2] J. Bannister and K. Trivedi, "Task Allocation in Fault Tolerant Distributed Systems," ACTA Informatica, Vol. 20, pp. 261-281, 1983.

[3] A. Bondavalli, F. Di Giandomenico and J. Xu, "A Cost Effective and Flexible Scheme for Software Fault Tolerance," Computing Laboratory, University of Newcastle upon Tyne, Tech. Report Series, no. 372, 1992.

[4] W. Burleson, J. Ko and et. al., "The Spring Scheduling Co-processor: a Scheduling Accelerator," in Proc. ICCD, Cambridge, Mass., USA, 1993.

[5] P. Ciompi, F. Grandoni and L. Strigini, "Software Fault Tolerance," in Delta-4: A Generic Architecture for Dependable Distributed Computing, D. Powell Ed., SpringerVerlag Research Reports ESPRIT, pp. 351-369, 1992.

[6] C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall, 1985.

[7] K. Kim and T. Lawrence, "Adaptive Fault Tolerance: Issues and Approaches," in Proc. Second IEEE Workshop on Future Trends of Distributed Computing Systems, Cairo, Egypt, pp. 38-46, 1990.

[8] C. Koza, "Scheduling of Hard Real-Time Tasks in the Fault Tolerant Distributed Real-Time System MARS," in Proc. 4th IEEE Workshop on Real-Time Operating Systems, Boston, Mass., pp. 31-36, 1987.

[9] C.M. Krishna and K. G. Shin, "On Scheduling Tasks With a Quick Recovery from Failures," IEEE TC, Vol. 35, pp. 448-455, 1986.

[10] A. R. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling Problem," IEEE TSE, Vol. SE-12, pp. 1089-1095, 1986.

[11] K. Lin, S. Natarajan and J. Liu, "Utilizing Partial Computations in Real-Time Systems," in Proc. IEEE Real-Time Systems Symposium, San Jose, California, pp. 210-215, 1987.

[12] C. Liu, "A General Framework for Software Fault Tolerance," ESPRIT BRA Project 3092 PDCS, Second Year Deliverables, 1991.

[13] J. Liu, K. Lin, W. Shih, J. Chung A. Yu and W. Zhao, "Algorithms for Scheduling Imprecise Computation," IEEE Computer, Vol. 24, pp. 58-67, 1991.

[14] D. Powell, G. Bonn, D. Seaton, P. Verissimo and F. Waeselynck, "The Delta4 Approach to Dependability in Open Distributed Computing Systems," in Proc. 18-th International Symposium on Fault Tolerant Computing, Tokyo, Japan, pp. 246-251, 1988.

[15] K. Ramamritham, "Scheduling Complex Periodic Tasks," in Proc. Intl. Conference on Distributed Computing Systems, 1990.

[16] C. Shen, K. Ramamritham and J. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," IEEE Transactions on Parallel and Distributed Computing, Vol. 4, No. 4, pp. 382-397, 1993.

[17] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," ACM Operating Systems Review, Vol. 23, pp. 54-71, 1989.

[18] J. Stankovic and K. Ramamritham, "What is Predictability for Real-Time Systems," Real-Time Systems Journal, Vol. 2, pp. 247-254, 1990.

[19] L. Strigini, "Software Fault Tolerance," ESPRIT BRA Project 3092 PDCS, Technical Report no. 23, 1990.

[20] G. F. Sullivan and G. M. Masson, "Using Certification Trails to Achieve Software Fault Tolerance," in Proc. 20th International Symposium on Fault-Tolerant Computing, Newcastle-upon-Tyne, U.K., pp. 423-431, 1990.

[21] J. Xu and D. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations in Real-Time Systems," IEEE TSE, Vol. 16, pp. 360-369, 1990.