(Paper Id: EMBS-116) Implementing Fault-Tolerance in Real-Time Systems by Program Transformations

Abstract

We present a formal approach to implement fault-tolerance in real-time embedded systems. The fault-intolerant initial system consists of a set of independent periodic tasks scheduled onto a set of fail-silent processors. We transform the tasks such that, assuming the availability of an additional spare processor, the system tolerates one failure at a time (transient or permanent). Failure detection is implemented using heartbeating, and failure masking using checkpointing and roll-back. These techniques are described and implemented by automatic program transformations on the tasks' programs. The proposed formal approach to fault-tolerance by program transformation highlights the benefits of separation of concerns and allows us to show whether the implementation satisfies real-time constraints.

I. Introduction

In most distributed embedded systems, such as automotive and avionics, fault-tolerance is a crucial issue [1], [2], [3]. Fault-tolerance is defined as the ability of the system to comply with its specification despite the presence of faults in any of its components [4]. To achieve this goal, we rely on two means: failure detection and failure masking. Among the two classes of faults, hardware and software, we only address the former. Tolerating hardware faults requires redundant hardware, be it explicitly added by the system's designer for this purpose, or intrinsically provided by the existing parallelism of the system. We assume that the system is equipped with one spare processor, which runs a special monitor module, in charge of detecting the failures in the other processors of the system, and then masking the failure.

We achieve failure detection thanks to timeouts; two popular approaches exist: the so-called "pull" and "push" methods [5]. In the pull method, the monitor sends liveness requests (*i.e.,* "are you alive?" messages) to the monitored components, and considers a component as faulty if it does not receive a reply from that component before a fixed time delay. In the push method, each component of the system periodically sends heartbeat information (*i.e.,* "I am alive" messages) to the monitor, which considers a component as faulty if two successive heartbeats are not received by the monitor within a predefined time interval [6]. We employ this last method which involves only one-way messages.

We implement failure masking with checkpointing and rollback mechanisms, which have been addressed in many works. It involves storing the global state of the system in a stable memory, and restoring the last state upon the detection of a failure to resume execution. There exist many implementation strategies of checkpointing and rollback, such as user-directed, compiler-assisted, system-level, library-supported, etc [7], [8], [9]. The pros and cons of these strategies are discussed in [10]. Checkpointing can be synchronous or asynchronous. In our setting where we consider only independent tasks, the simplest approach is asynchronous checkpointing. Tasks take local checkpoints periodically without any coordination with each other. This approach allows maximum component autonomy for taking checkpoints and has no message overhead.

We propose a framework based on automatic program transformations to implement fault-tolerance in distributed embedded systems. Our starting point is a fault-intolerant system, consisting of a set of independent periodic hard real-time tasks scheduled onto a set of fail-silent processors. The goal of the transformations is to obtain a system tolerant to one hardware failure. One spare processor is initially free of tasks: it will run a special monitor task, in charge of detecting and masking the system's failures. Each transformation will implement a portion of either the detection or the masking of failures. For instance,

one transformation will add the checkpointing code into the real-time tasks, while another one will add the rollback code into the monitor task. The transformations will be guided by the fault-tolerance properties required by the user. Our assumption that all tasks are independent (*i.e.,* they do not communicate with each other) simplifies the problem of consistent global checkpointing, since all local checkpoints belong to the set of global consistent checkpoints.

One important point of our framework is the ability to formally prove that the transformed system satisfies the real-time constraints even in the presence of one failure. The techniques that we present (checkpointing, rollback, heartbeating, etc) are pretty standard in the OS context. Our contribution is to study them in the context of hard real-time tasks, to express them formally as automatic program transformations, and to prove formal properties of the system after the transformations.

Section II gives an overview of our approach. In Section III, we give a formal definition for the real-time tasks and we introduce a simple programming language. Section IV presents program transformations implementing checkpointing and heartbeating. We present the monitor task in Section V and extend our approach to transient and multiple failures in Section VI. Finally, we review related work in Section VII and conclude in Section VIII.

II. Overview of the Proposed System

We consider a distributed embedded system consisting of p processors plus a spare processor, a stable memory, and I/O devices. All are connected via a communication network (see Figure 1). We assume a reliable communication and a deterministic transmission time (0 units of time for the sake of clarity) between the processors.

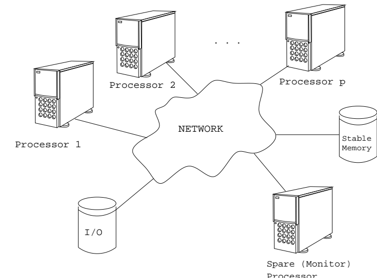
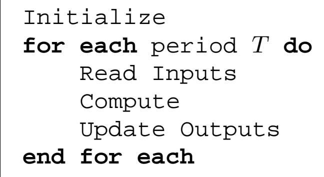


Fig. 1: System architecture.

Our failure assumption is that all the processors show omission/crash failure behavior [3]. This means that the processors may transiently or permanently stop responding, but do not pollute the healthy remaining ones.

The system also has n real-time tasks that fit the simple-task model of [11]: all tasks are periodic and independent (i.e., no precedence constraints). More precisely, the program of each task has the form described in Figure 2.



```
Initialize
for each period T do
    Read Inputs
    Compute
    Update Outputs
end for each
```

Fig. 2: Program model of periodic real-time tasks.

We do not address the issue of distribution and scheduling of the tasks onto the processors. Hence, for the sake of clarity, we assume that each processor runs one single task (i.e., $n = p$). Executing more than one task on each processor (e.g., with a multi-rate cyclic execution approach) is still possible however.

Our approach deals with the programs of the tasks and defines program transformations on them to achieve fault-tolerance. We consider programs in compiled form at the assembly or binary code level, which allows us to evaluate WCET. We represent these three-address programs using a small imperative language. Since the system contains only one redundant processor, we provide a masking of only one processor failure at a time. Masking of more than one transient processor failure at a time could be achieved with additional spare processors (see Section VI).

The stable memory is used to keep the global state. The global state provides masking of processor failures by rolling-back to this safe state as soon as a failure is detected. The stable memory also stores one shared variable per processor, used for failure detection: the program of each task, after transformation, will periodically write a `1` into this shared variable, while the monitor will periodically (and with the same period) check that its value is indeed `1` and will reset it to `0`. When a failure occurs, the shared variable corresponding to the faulty processor will remain equal to `0`, therefore allowing the monitor

to detect the failure. The spare processor provides the necessary hardware redundancy and executes the monitor program for failure detection and masking purposes.

When the monitor detects a processor failure, it rolls back to the latest local state of the faulty processor stored in the stable memory. Then, it resumes the execution of the task that was running on the faulty processor, from this local state. Remember that, since the tasks are independent, the other tasks do not need to roll back to their own previous local state. This failure masking process is implemented by an asynchronous checkpointing, i.e., processors take local checkpoints periodically without any coordination with each other.

The two program transformations used for adding periodic heartbeating / failure detection and periodic checkpointing / rollback amounts to inserting code at specific points. This process may seem easy, but the conditional statements of the program to be transformed, e.g., `if` branchings and `while` loops, create many different execution paths, making it actually quite difficult. We therefore propose a preliminary program transformation, which equalizes the execution times between all the possible execution paths. This is done by padding dummy code in `if` branchings and by transforming `while` loops into `for` loops. After this transformation, the resulting programs have a constant execution time. Then, checkpointing and heartbeating commands are inserted in the code at constant time intervals. The periods between checkpoints and heartbeats are chosen in order to minimize their cost while satisfying the real-time constraints. A special monitoring program is also generated from the parameters of these transformations. The monitor consists of a number of tasks that must be scheduled by an algorithm providing deadline guarantees.

The algorithmic complexity of our program transformations is linear in the size of the program. The overhead in the transformed program is due to the fault-tolerance techniques we use (heartbeating, checkpointing and rollback). This overhead is unavoidable and compares favorably to the overhead induced by other fault-tolerance techniques, e.g., hardware and software redundancy.

III. TASKS

A real-time periodic task $\tau = (S,T)$ is specified by a program S and a period T. The program S is repeatedly executed each T units of time. A program usually reads its inputs (which are stored in a local variable), executes some statements, and writes its outputs (see Figure 2). Each task also has a deadline $d \leq T$ that it must satisfy when writing its output. To simplify the presentation, we take the deadline equal to the period but our approach does not depend on this assumption. Hence, the real-time constraint associated to the task (S,T) is that its program S must terminate before the end of its period T.

Programs are expressed in the following while-loop language:

$$
\begin{array}{lll}
S ::= & x{:=}A & \textit{assignment} \\
\mid & \texttt{skip} & \textit{no operation} \\
\mid & \texttt{read}(i) & \textit{input read} \\
\mid & \texttt{write}(o) & \textit{output write} \\
\mid & S_1;S_2 & \textit{sequencing} \\
\mid & \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 & \textit{conditional} \\
\mid & \texttt{while } B \texttt{ do } S & \textit{iteration}
\end{array}
$$

where A and B represent integer expressions (arithmetic expressions on integer variables) and boolean expressions (comparisons, `and`, `not`, etc) respectively. Here, we assume that the only variables used to store inputs and outputs are i and o. These instructions could be generalized to multiple reads and writes or to IO operations parameterized with a port. This language is well-known, simple and expressive enough. The reader may refer to [12] for a complete description.

The following example program *Fac* reads an unsigned integer variable and places it in i. It bounds the variable i by 10 and calculates the factorial of i which is finally written as output.

$$
\begin{aligned}
Fac \;=\; & \texttt{read}(i)\ ;\\
& \texttt{if}\ i > 10\ \texttt{then}\ i := 10;\ o := 1;\ \texttt{else}\ o := 1;\\
& \texttt{while}\ i > 0\ \texttt{do}\ \ o := o * i;\\
& \qquad\qquad\qquad\ i := i - 1;\\
& \texttt{write}(o);
\end{aligned}
$$

The simplest statement of the language is `skip` (the *nop* instruction), which exists on all processors. We take the execution time of the `skip` command to be the unit of time and we assume that the execution times of all other statements are multiple of the execution time of `skip`. A more fundamental assumption is that the *worst case execution time* (WCET) of any statement (or expression) S can be evaluated. The WCET analysis is the topic of much work (see [13], [14] for instance); We shall not dwell upon this issue any further.

For the remaining of the article, we fix the WCET of statements to be:

$$
\begin{array}{lll}
\text{WCET}(\texttt{skip}) = 1 & \text{WCET}(\texttt{read}) = 3 & \text{WCET}(\texttt{write}) = 3\\
\text{WCET}(S_1;S_2) = \text{WCET}(S_1) + \text{WCET}(S_2) & & \text{WCET}(x := e) = 3\\
\multicolumn{3}{l}{\text{WCET}(\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = 1 + max(\text{WCET}(S_1), \text{WCET}(S_2))}\\
\multicolumn{3}{l}{\text{WCET}(\texttt{while}\ B\ \texttt{do}\ S) = \text{WCLC}(\texttt{while}\ B\ \texttt{do}\ S) \times (\text{WCET}(S) + 1) + 1}
\end{array}
$$

for any "simple" expressions e or b. Using temporary variables, it is always possible to split complex arithmetic and boolean expressions so that they remain simple enough (as in three-address code). Note that it is necessary to know the *worst-case loop count* (WCLC) of each loop to compute the WCET of a program with loops. Inversely, we have:

$$
\text{WCLC}(\texttt{while}\ B\ \texttt{do}\ S) = \frac{\text{WCET}(\texttt{while}\ B\ \texttt{do}\ S) - 1}{\text{WCET}(S) + 1}
$$

where the "+1" accounts for the test of B done inside each loop, and the "-1" accounts for the fact that the last test of B (the one that evaluates to `false`) terminates the loop.

With these figures, we get $\text{WCET}(Fac) = 84$. In the rest of the article, we consider the task $(Fac, 200)$, that is to say Fac with a deadline/period of 200 time units.

The real-time property for a system of n tasks $\{(S_1, T_1), \ldots, (S_n, T_n)\}$ is that each task must meet its deadline. Since each processor runs a single task, it amounts to:

$$
\forall i \in \{1, 2, ..., n\}, \text{WCET}(S_i) \leq T_i \tag{1}
$$

The semantics of a statement S is given by the function $[\![S]\!] : \mathbf{State} \rightarrow \mathbf{State}$. A state $s \in \mathbf{State}$ maps program variables $\mathcal{V}$ to their values. The semantic function takes a statement S, an initial state s_0 and yields the resulting state s_f obtained after the execution of the statement: $[\![S]\!]s_0 = s_f$. Several equivalent formal definitions of $[\![.]\!]$ (operational, denotational, axiomatic) can be found in [12].

The IO semantics of a task (S, T) is given by a pair of streams

$$
(i_1, \ldots, i_n, \ldots), (o_1, \ldots, o_n, \ldots)
$$

where i_k is the input provided by the environment during the kth period and o_k is the last output written during the kth period. So, if several $write(o)$ are performed during a period, the semantics and the environment will consider only the last one. We also assume that the environment proposes the same input during a period: several $read(i)$ during the same period will result in the same readings.

For example, if the environment proposes 2 as input then the program

$$
\texttt{read}(i);\ o := i;\ \texttt{write}(o);\ \texttt{read}(i);\ o := o * i;\ \texttt{write}(o)
$$

produces 4 as output during that same period, and not $(2, 4)$. Assuming $\mathbb{N}$ as inputs, the IO semantics of Fac is:

$$
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \ldots), (0, 1!, 2!, 3!, 4!, 5!, 6!, 7!, 8!, 9!, 10!, 10!, 10!, \ldots)
$$

IV. PROGRAM TRANSFORMATIONS

Failure detection and failure masking rely on inserting heartbeating and checkpointing instructions in programs. These instructions must be inserted such that they are executed periodically. We therefore transform a task program such that an heartbeat and a checkpoint are executed every T_{HB} and T_{CP} period of time respectively. Conditional statements (`if` and `while`) complicate this insertion. They both lead to many paths with different execution times. It is therefore impossible to insert instructions at constant time intervals without duplicating the code. To avoid this problem, we first transform the program in order to fix the execution time of all conditional and loops to their worst case execution time. Intuitively, it amounts to adding dummy code to conditional and loop statements. Such transformations suppose to be able to evaluate the WCET of programs, and in particular, the WCLC for every loop. After this time equalization, checkpoints and heartbeats can be introduced simply using the same transformation. To ensure timing correctness in these transformations, we also assume the use of processors that do not have complex behaviors such as pipelining and caching. This assumption is realistic since the real-time embedded applications generally do not rely on these features.

A transformation may increase the WCET of programs. So, after each transformation $\mathcal{T}$, the real-time constraint $\text{WCET}(\mathcal{T}(S)) \leq T$ must be checked; thanks to our assumptions on WCET this can be done automatically.

A. Equalizing Execution Time

Equalizing the execution time of a program consists of two basic transformations:

- *Branch transformation*: the execution times of the two branches of a conditional are equalized by padding dummy code in the less expensive branch.
- *Loop transformation*: each `while` loop is transformed into a loop with a constant iteration count (*i.e.,* a `for` loop). The iteration count of the new loop is set to the WCLC of the original loop. This technique is entirely the same with the loop transformation used in “single path programming” [15].

The dummy code added for padding is sequences of `skip` statements. We write $\texttt{skip}^n$ to represent a sequence of n `skip` statements: $\text{WCET}(\texttt{skip}^n) = n$. In order to alleviate the notation, we introduce `for` loops:

$$\texttt{for } l := 1 \texttt{ to } n \texttt{ do } S$$

with $\text{WCET}(\texttt{for } l := 1 \texttt{ to } n \texttt{ do } S) = n \times \text{WCET}(S) + 1$. The global equalization process is defined inductively by the following transformation, noted $\mathcal{F}$. The rules below must be understood like a case expression in the programming language ML [16]: cases are evaluated from top to bottom, and the transformation rule corresponding to the first pattern that matches the input program is performed.

$$\begin{array}{lcl}
\mathcal{F}[\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2] & = & \texttt{if } B \texttt{ then } \mathcal{F}[S_1];\ \texttt{skip}^{\max(0,\delta_2-\delta_1)}; \\
 & & \quad\quad \texttt{else } \mathcal{F}[S_2];\ \texttt{skip}^{\max(0,\delta_1-\delta_2)}; \\
 & & \text{with } \ \delta_i = \text{WCET}(\mathcal{F}[S_i]) \ \text{ for } i = 1,2 \\
\mathcal{F}[\texttt{while } B \texttt{ do } S] & = & \texttt{for } l = 1 \texttt{ to } \text{WCLC}(\texttt{while } B \texttt{ do } S) \texttt{ do} \\
 & & \texttt{if } B \texttt{ then } \mathcal{F}[S] \texttt{ else } \texttt{skip}^{\text{WCET}(\mathcal{F}[S])}; \\
\mathcal{F}[S_1;S_2] & = & \mathcal{F}[S_1];\ \mathcal{F}[S_2] \\
\mathcal{F}[S] & = & S \quad\quad \text{otherwise}
\end{array}$$

While loops and conditionals are the only statements subject to code modification. The branch transformation adds as many `skip` as needed to match the execution time of the other branch. The most expensive branch remains unchanged. The transformation is applied inductively to the statement of each branch prior to this equalization.

The loop transformation uses a temporary variable (l, assumed to be fresh) to implement a constant iteration count loop. After transformation, the body of the loop will always be executed as many times

as the worst case loop count. The body of the loop is made of an (equalized) conditional to prevent the execution of the body S when B has become false. As in the branch transformation, inner statements are transformed first.

It is easy to show that for any program S, the best and worst case execution times of $\mathcal{F}[S]$ are the same. Furthermore, the transformation $\mathcal{F}$ does not change the WCET of programs:

Property 1 $\forall S, \quad \text{BCET}(\mathcal{F}[S]) = \text{WCET}(\mathcal{F}[S])$

Property 2 $\forall S, \quad \text{WCET}(S) = \text{WCET}(\mathcal{F}[S])$

These two properties are easily proved by induction on the transformation rules. However, property 2 depends on the assumptions we made on the respective WCET of `while` loops and `for` loops. If the branch transformation clearly leaves the WCET unchanged, the loop transformation may add a small overhead due to the counting of l. In any case, the WCET of source and transformed programs remain close to each other.

The transformation applied on our example Fac produces:

$$
\begin{array}{rl}
Fac_1 = \mathcal{F}[Fac] = & \texttt{read}(i); \\
& \texttt{if } i > 10 \texttt{ then } i := 10;\ o := 1;\ \texttt{else } o := 1;\ \texttt{skip}^3; \\
& \texttt{for } \ l = 1 \texttt{ to } 10 \texttt{ do} \\
& \quad\quad \texttt{if } i > 0 \ \texttt{ then } i := i - 1;\ o := o * i;\ \texttt{else skip}^6; \\
& \texttt{write}(o);
\end{array}
$$

B. Checkpointing and Heartbeating

Checkpointing and heartbeating both involve the insertion of special commands at appropriate program points. We introduce two new commands:

- `hbeat` sends an heartbeat telling the monitor that the processor is alive. This command is implemented by setting a special variable in the stable memory. The vector $\text{HBT}[1 \ldots n]$ gathers the heartbeat variables of the n tasks. The command `hbeat` in task i is implemented as $\text{HBT}[i] := 1$.
- `checkpt` saves the current state in the stable memory. It is sufficient to save only the live variables and only those which have been modified since the last checkpoint. This information can be inferred by static analysis techniques. Here, we simply assume that `checkpt` saves enough variables to revert to a valid state when needed.

Heartbeating is usually done periodically, whereas the policies for checkpointing differ. Here, we chose periodic heartbeats and checkpoints. In our context, the key property is to meet the real-time constraints. We will see in section V how to compute the optimal periods for those two commands, optimality being defined *w.r.t.* those real-time constraints.

In this section, we define a transformation $\mathcal{I}_c^T(S, t)$ that inserts the command c every T units of time in the program S. It will be used both for checkpointing and heartbeating. The parameter T denotes the period whereas the time counter t counts the time remaining before the next insertion. Of course, inserting the command c must not break atomic statements. So, the time between insertions cannot be exactly T, but the delay will be tightly bounded. The transformation $\mathcal{I}$ returns a pair consisting of the transformed program and the number of time units before the next insertion. It relies on the property that all paths of the program have the same execution time (see Section IV-A). In order to insert heartbeats afterwards, this property should remain valid after the insertion of checkpoints. We may either assume that `checkpt` takes the same time when inserted in different paths (*e.g.*, the two branches of a conditional),

or re-apply the transformation $\mathcal{F}$ after checkpointing. Again, the rules below must be understood like a case expression in ML.

$$\mathcal{I}_c^T(S,t) = (S, t - \text{WCET}(S)) \qquad \text{if WCET}(S) < t$$

$$\mathcal{I}_c^T(S,t) = \mathcal{I}_c^T(c;S,T) \qquad \text{if } t < 0$$

$$\mathcal{I}_c^T(a,t) = (a;c, T - \text{WCET}(c)) \qquad \text{with } a \text{ an atomic command}$$

$$\mathcal{I}_c^T(S_1;S_2,t) = (S_1';S_2',t_2) \qquad \text{with } (S_1',t_1) = \mathcal{I}_c^T(S_1,t) \text{ and } (S_2',t_2) = \mathcal{I}_c^T(S_1,t_1)$$

$$\mathcal{I}_c^T(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, t) = (\texttt{if } b \texttt{ then } S_1' \texttt{ else } S_2', max(t_1,t_2)) \qquad \text{with } (S_1',t_1) = \mathcal{I}_c^T(S_1,t-\delta) \text{ and } (S_2',t_2) = \mathcal{I}_c^T(S_2,t-\delta) \text{ and } \delta \text{ is the WCET of the test/jump}$$

$$\mathcal{I}_c^T(\texttt{for } l = 1 \texttt{ to } n \texttt{ do } S, t) = (Fold(S'), t') \qquad \text{with } (S',t') = \mathcal{I}_c^T(S^n,t))$$

When the statement S finishes before the next insertion time t, the transformation terminates. In all the other cases, the WCET of S is greater then t and at least one insertion can be done.

The second rule applies when the time counter is negative. This case may arise when the ideal time for inserting the command c is in the middle of the boolean expression of a conditional. When t is negative, the command must be inserted right away. The transformation proceeds with the resulting program, with t reset to T. Hence, there is a potential drift in the clock of heartbeats but our technique tolerates it.

When the program is an atomic command a (whose execution time is greater than or equal to t), the command c is inserted right after a. Even though c has been inserted $(\text{WCET}(a) - t)$ units of time later than the ideal point, the time target for the next insertion is reset to $T - \text{WCET}(c)$ (again, a potential clock drift).

The insertion in a sequence $S_1;S_2$ is first done in S_1, which also returns the time remaining before the next insertion in S_2.

For conditional statements, the insertion is performed in both branches. The time of the test and branching is taken into account via the δ parameter (equal to 1 for simple tests and in our example). If the statement S has been previously transformed by the time equalization, then both branches have the same WCET. After the insertion, the difference between the time counters t_1 and t_2 is at most the WCET of the most expensive atomic instruction.

The rule for loops unrolls them completely, performs the insertion, and then factorizes code by folding code in `for` loops as much as possible. We do not describe the *Fold* function in details; the following two rules should provide some intuition:

$$Fold(S;S) = \texttt{for } l = 1 \texttt{ to } 2 \texttt{ do } S$$
$$Fold((\texttt{for } l = 1 \texttt{ to } k \texttt{ do } S);S) = \texttt{for } l = 1 \texttt{ to } k+1 \texttt{ do } S$$

Actually, it would be possible to express the transformation $\mathcal{I}$ such that it minimally unrolls loops and does not need folding. However, the rules are much more complex to present.

The transformation assumes that the period is greater than the cost of the command, *i.e.*, $T > \text{WCET}(c)$. Otherwise, the insertion may loop by inserting c within c and so on.

Property 3 *In a transformed program $\mathcal{I}_c^T(S,T)$, the actual time interval Δ between the beginning of two successive commands c is such that:*

$$T \leq \Delta < T + \varepsilon$$

with ε being the WCET of the most expensive atomic instruction (assignment or test) in the program.

To prove this property, we must first formalize execution of programs (*e.g.*, using a small step operational semantics). Then, Δ can be defined properly on the execution traces and the property proved by induction. Such developments are beyond the scope of this paper.

Checkpointing and heartbeating are performed using the transformation $\mathcal{I}$. Checkpoints are inserted first and heartbeats last. The period between two checkpoints must take into account the overhead that will be added by heartbeats afterward. The overhead added by heartbeating during X units of time is $\frac{X\overline{h}}{T_{HB}-\overline{h}}$ with $\overline{h} = \text{WCET}(\texttt{hbeat})$. So, if T_{CP} is the desired period of checkpoints, we must use the period T'_{CP} defined by the equation:

$$T'_{CP} = T_{CP} - \frac{T'_{CP}\overline{h}}{T_{HB} - \overline{h}} \iff T'_{CP}\left(1 + \frac{\overline{h}}{T_{HB} - \overline{h}}\right) = T_{CP} \iff T'_{CP} = \frac{T_{CP}}{1 + \frac{\overline{h}}{T_{HB}-\overline{h}}} \quad (2)$$

With these notations, the insertion of checkpoints and heartbeats is described by the following ML code:

$$\begin{array}{lll} \text{let} & (S', -) = \mathcal{I}^{T'_{CP}}_{\texttt{checkpt}}(S, T'_{CP}) & \text{in} \\ \text{let} & ((S'';\texttt{hbeat}), -) = \mathcal{I}^{T_{HB}}_{\texttt{hbeat}}(\texttt{hbeat};\ S', T_{HB}) & \text{in} \\ & S'';\texttt{hbeat}(k) & \end{array}$$

A first heartbeat is added right at the beginning of S', the other heartbeats are inserted by $\mathcal{I}$, then last heartbeat is replaced by $\texttt{hbeat}(k)$. We can always ensure that S' finishes with a heartbeat by padding dummy code at the end. The command $\texttt{hbeat}(k)$ is a special heartbeat that sets the variable to k instead of 1, *i.e.*, $\text{HBT}[i] := k$. Following this last heartbeat, the monitor will therefore decrease the shared variable and will resume error detection when the variable becomes 0 again. This mechanism accounts for the idle interval of time between the termination of S'' and the beginning of the next period. Hence, k has to be computed as:

$$k = \left\lceil \frac{T - \text{WCET}(S'';\texttt{hbeat})}{T_{HB}} \right\rceil \quad (3)$$

After the introduction of heartbeats, the period between checkpoints will be $T'_{CP}\left(1 + \frac{\overline{h}}{T_{HB}-\overline{h}}\right)$, *i.e.*, T_{CP}. More precisely, it follows from Property 3 that:

Property 4 *The actual time intervals Δ_{CP} and Δ_{HB} between two successive checkpoints and heartbeats are such that:*

$$T_{CP} \leq \Delta_{CP} < T_{CP} + \varepsilon + \overline{h} \qquad \textit{and} \qquad T_{HB} \leq \Delta_{HB} < T_{HB} + \varepsilon$$

As pointed out above, the transformation $\mathcal{I}$ requires the period to be bigger than the cost of the command. For checkpointing and heartbeating we must ensure that:

$$T'_{CP} > \text{WCET}(\texttt{hbeat}) \qquad \text{and} \qquad T_{HB} > \text{WCET}(\texttt{checkpt})$$

To illustrate these transformations on our previous example, we take:

$$\text{WCET}(\texttt{hbeat}) = 3 \quad \text{WCET}(\texttt{checkpt}) = 10 \quad T_{CP} = 80 \quad T_{HB} = 10$$

So, we get $T'_{CP} = 80 - \frac{3*T'_{CP}}{10-3}$ *i.e.*, $T'_{CP} = 56$ and $\mathcal{I}^{56}_{\texttt{checkpt}}(Fac_1, 56)$ produces:

$$\begin{array}{lll} Fac_2 & = & \texttt{read}(i); \\ & & \texttt{if}\ i > 10\ \texttt{then}\ i := 10;\ o := 1;\ \texttt{else}\ o := 1;\ \texttt{skip}^3; \\ & & \texttt{for}\ \ l = 1\ \texttt{to}\ 6\ \texttt{do} \\ & & \qquad \texttt{if}\ i > 0\ \ \texttt{then}\ i := i - 1; o := o * i;\ \texttt{else skip}^6; \\ & & \texttt{if}\ i > 0\ \ \texttt{then}\ i := i - 1;\ \texttt{checkpt};\ o := o * i;\ \texttt{else skip}^2;\ \texttt{checkpt};\ \texttt{skip}^4; \\ & & \texttt{for}\ \ l = 1\ \texttt{to}\ 3\ \texttt{do} \end{array}$$

$\quad\quad$ if $i > 0$ then $i := i - 1; o := o * i$; else skip^6;
write(o);

A single checkpt is inserted after 56 time units, which happens within the conditional of the 7th iteration of the for loop. The checkpoint is inserted with one time unit delay in the then branch and right on time in the else branch. The transformation proceeds by unrolling the loop, inserting checkpt at the right places. Portions of the code are then refactorized in two for loops.

For the next step, we suppose, for the sake of the example, that checkpt, which takes 10 units of time, can be split in two parts checkpt = checkpt$_1$;checkpt$_2$ where checkpt$_1$ takes 3 time units and checkpt$_2$ takes 7. We add an heartbeat as a first instruction and, in order to finish with an heartbeat, we must add 4 skip at the end. The transformation $\mathcal{I}^{10}_{\texttt{hbeat}}(Fac_2, 10)$ inserts an heartbeat every 10 time units and yields:

Fac_3 = hbeat; read(i);
if $i > 10$ then $i := 10$; hbeat; $o := 1$; else $o := 1$; hbeat; skip^3;
for $l = 1$ to 6 do
$\quad$ if $i > 0$ then $i := i - 1$; hbeat; $o := o * i$; else skip^3; hbeat; skip^3;
if $i > 0$ then $i := i - 1$; checkpt$_1$; hbeat; checkpt$_2$; hbeat; $o := o * i$;
$\quad\quad$ else skip^2; checkpt$_1$; hbeat; checkpt$_2$; hbeat; skip^4;
for $l = 1$ to 3 do
$\quad$ if $i > 0$ then $i := i - 1; o := o * i$; else skip^6;
$\quad$ hbeat;
write(o); skip^4; hbeat;

Note that in Fac_3, the checkpoint is performed exactly after 80 units of time, as needed. Finally, since WCET(Fac_3) = 141 and the period is 200, equation (3) gives $\lceil \frac{200-141}{10} \rceil = 6$, so the last hbeat must be changed into hbeat(6). Figure 3 illustrates the form of a general program (*i.e.*, not Fac_3) after all the transformations:

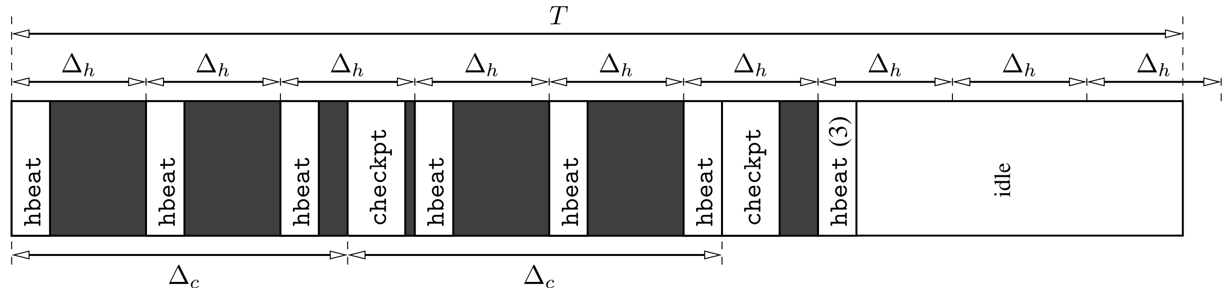


Fig. 3. Program with checkpointing and heartbeating.

V. Implementing the Monitor

A special program called *monitor* is executed on the spare processor. As already explained, the monitor performs failure detection by checking the heartbeats sent by each other task. The other responsibility of the monitor is to perform a roll-back recovery in case of a failure. In our case, roll-back recovery involves restarting the failed task on the spare processor from its latest state stored in the stable memory. In the following subsections, we comprehensively explain heartbeat detection and roll-back recovery actions, together with the implementation details and conditions for real-time guarantee.

A. Failure Detection

The monitor periodically checks the heartbeat variables HBT[i] to be sure of the liveness of the processor running the tasks τ_i. For a correct operation and fast detection, it must check each HBT[i] at least at the period T_{HB_i}. Since each processor (or each task) has a potentially different heartbeat period by construction, the monitor should concurrently check all the variables at their own speed. A common solution to this problem is to schedule one periodic task for each of the n other processors. The period of the task is equal to the corresponding heartbeating interval. Therefore, the monitor has n real-time periodic tasks $\Gamma_i = (Det_i, T_{HB_i})$, with $1 \leq i \leq n$, plus one aperiodic recovery task that will be explained later. The deadline of each task Γ_i is equal to its period. The program Det_i is:

$$Det_i \;=\; \texttt{HBT}[i] := \texttt{HBT}[i] - 1;\;\; \texttt{if}\ x = -2\ \texttt{then run}\ Rec(i);$$

When it is positive, HBT[i] contains the number of T_{HB_i} periods before the next heartbeat of τ_i, hence the next update of HBT[i]. When it is equal to -2, the monitors decides that the processor i is faulty, so it must launch the failure recovery program Rec. When it is equal to -1, the processor i is suspect but not declared faulty. Indeed, it might just be late, or HBT[i] might not have been updated yet due to the clock drift between the two processors.

In order to guarantee the real-time constraints, we must compute the worst case failure detection time α_i for each task τ_i. The detector is not synchronized with the tasks, therefore the heartbeat send times (σ_k) of τ_i and the heartbeat check times (σ'_k) of Det_i may differ such that $|\sigma_k - \sigma'_k| < T_{HB_i}$. In the worst case, *i.e.*, if $\sigma_k - \sigma'_k \simeq T_{HB_i}$ and τ_i has failed right after sending a heartbeat, the detector can see this heartbeat one period later and becomes suspect. It detects the failure at the end of this period. Note that the program transformation always guarantees the interval between two consecutive heartbeats to be within $[T_{HB_i}, T_{HB_i} + \varepsilon)$.

Let L_r and L_w denote respectively the times necessary for reading and writing a heartbeat variable, let ξ_i be the maximum time drift between Det_i and τ_i within one heartbeat interval ($\xi_i \ll T_{HB_i}$), then the worst case detection time α_i of the failure of task τ_i satisfies:

$$\alpha_i < 3(T_{HB_i} + \varepsilon) + L_r + L_w + 3\xi_i \tag{4}$$

Finally, the problem of the clock drift between the task τ_i that writes HBT[i] and the task Det_i that reads HBT[i] must be addressed. Those two tasks have the same period T_{HB_i}, but since the clocks of the two processors are not synchronized, there are drifts. We assume that these clocks are *quasi-synchronous* [17], meaning that any of the two clocks cannot take the value `true` more than twice between two successive `true` values of the other one. This is the case in many embedded architectures (e.g. TTA and FlexRay for automotive) [18]. With this hypothesis, τ_i can write HBT[i] twice in a row, which is not a problem. Similarly, Det_i can read and decrement HBT[i] twice in a row, again which is not a problem since Det_i decides that τ_i is faulty only after three successive decrements (*i.e.*, from 1 to -2).

B. Roll-back Recovery

As soon as the monitor detects a processor failure, it restarts the failed task from the latest checkpoint. This means that the monitor does not exist anymore since the spare processor stops the monitor and starts executing the failed task instead. The following program represents the recovery operation:

$$Rec\ (\mathrm{x}) = \mathrm{FAILED}{:=}x;\quad \texttt{restart}\ (\tau_x,\ \mathrm{CONTEXT}_x);$$

where `restart` (τ_x, CONTEXT$_x$) is a macro that stops the monitor application and instead restarts τ_x from its latest safe point specified by CONTEXT$_x$. The shared variable FAILED holds the identification number

of the failed task. FAILED $= 0$ indicates that there is no failed processor. FAILED $= x \in \{1, 2, \ldots, n\}$ indicates that τ_x has failed and has been restarted on the spare processor. The recovery time (denoted with β) after a failure occurrence can be defined as the sum of the failure detection time plus the time to re-execute the part of the code after the last checkpoint. If we denote the time for context reading by L_C, then the worst case recovery time is:

$$\beta = 3(T_{HB} + \varepsilon) + T_{CP} + L_r + L_w + L_C + 3 \max_{1 \leq i \leq n} \xi_i + \text{WCET}(Det) + \text{WCET}(Rec) \quad (5)$$

C. Satisfying the Real-Time Constraints

After the program transformations, the WCET of the fault-tolerant program of the task (S'', T), taking into account the recovery time, is given by the following expression:

$$\text{WCET}(S'') = \text{WCET}(S) + \left\lfloor \frac{\text{WCET}(S)}{T_{CP}} \right\rfloor \times \text{WCET}(\texttt{checkpt}) + \left(\frac{\text{WCET}(S)}{T_{HB}} + 1 \right) \times \text{WCET}(\texttt{hbeat}) + \beta \quad (6)$$

Note that this WCET includes both the error detection time and recovery time. We are interested in the *optimum* values, $T^\star_{CP}$ and $T^\star_{HB}$, *i.e.*, the values that offer the best trade-off between fast failure detection, fast failure recovery, and least overhead due to the code insertion. If we combine the expression (6) and the equation (5), we obtain a two-value function f of the form:

$$f(T_{CP}, T_{HB}) = \frac{\text{WCET}(S) \times \text{WCET}(\texttt{checkpt})}{T_{CP}} + \frac{\text{WCET}(S) \times \text{WCET}(\texttt{hbeat})}{T_{HB}} + 3T_{HB} + T_{CP} + K \quad (7)$$

where K is a constant. Note that neglecting the floor function in (6) is for the purpose of an explicit calculation and causes approximate results.

Since the least overhead due to the code insertion means the smallest WCET for S'', we have to minimize f. Now, the computation of its two partial derivatives yields:

$$\frac{\partial f}{\partial T_{CP}} = 1 - \frac{\text{WCET}(S) \times \text{WCET}(\texttt{checkpt})}{T^2_{CP}} \quad \text{and} \quad \frac{\partial f}{\partial T_{HB}} = 3 - \frac{\text{WCET}(S) \times \text{WCET}(\texttt{hbeat})}{T^2_{HB}} \quad (8)$$

Since the two second partial derivatives are positive in the $(0, +\infty) \times (0, +\infty)$ portion of the space, the function f is *convex* and the optimal values $T^\star_{CP}$ and $T^\star_{HB}$ are those that nullify the two first order partial derivatives. Hence:

$$\left. \frac{\partial f}{\partial T_{CP}} \right|_{T_{CP} = T^\star_{CP}} = 0 \implies T^\star_{CP} = \sqrt{\text{WCET}(S) \times \text{WCET}(\texttt{checkpt})} \quad (9)$$

$$\left. \frac{\partial f}{\partial T_{HB}} \right|_{T_{HB} = T^\star_{HB}} = 0 \implies T^\star_{HB} = \sqrt{\frac{1}{3} \text{WCET}(S) \times \text{WCET}(\texttt{hbeat})} \quad (10)$$

With our *Fac* example, we get $T^\star_{CP} = \sqrt{84 \times 10} \simeq 28.98$ and $T^\star_{HB} = \sqrt{\frac{84 \times 3}{3}} \simeq 9.16$. This means that the values we have chosen, respectively 80 and 10, were not the optimal values.

Equations (9) and (10) give the optimal values for the heartbeat and checkpoint periods. Finally, in order to satisfy the real-time property of the whole system, the only criterion that should be checked is:

$$f(T_{CP_i}, T_{HB_i}) < T_i, \qquad \forall i \in \{1, 2, \ldots, n\} \quad (11)$$

Removing the assumption of zero communication time just involves adding a worst case communication delay parameter in formulae (4) and (5), which does not have an effect on the optimum values, $T^\star_{CP}$ and $T^\star_{HB}$.

D. Scheduling all the Detection Tasks

The monitoring application consists of n detector tasks plus one recovery task. Detector tasks are periodic and independent, whereas the recovery task is will be executed exactly once, at the end of the monitoring application (when a failure is detected). Therefore, it can be disregarded in the schedulability analysis. We thus have the task set $\Gamma = \{(Det_1, T_{HB_1}),\ (Det_2, T_{HB_2}),\ \ldots,\ (Det_n, T_{HB_n})\}$ that must satisfy:

$$\forall i \in \{1, 2, ..., n\}, \text{WCET}(Det_i) \leq T_{HB_i}. \tag{12}$$

Preemptive scheduling techniques such as Rate-Monotonic (RM) and Earliest-Deadline-First (EDF) settle the problem. Both RM and EDF are the major paradigms of preemptive scheduling, and basic schedulability conditions for them were derived by Liu and Layland for a set of n periodic tasks under the assumptions that all tasks start at time $t = 0$, relative deadlines are equal to their periods and tasks are independent [19]. RM is a fixed priority based preemptive scheduling where tasks are assigned priorities inversely proportional to their periods. In EDF, however, priorities are dynamically assigned inversely proportional to the tasks' deadlines (in other words, as a task becomes nearer to its deadline, its priority increases). For many reasons, as remarked in [20], RM is the most common scheduler implemented in commercial RTOS kernels. In our context, it guarantees that Γ is schedulable if:

$$\sum_{i=1}^{n} \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 2(2^{1/n} - 1) \tag{13}$$

Under the same assumptions, EDF guarantees that Γ is schedulable if:

$$\sum_{i=1}^{n} \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 1 \tag{14}$$

The above schedulability conditions highlight that EDF allows a better processor utilization while both are appropriate and sufficient for scheduling the monitoring tasks with deadline guarantee.

VI. Tolerating Transient and Multiple Failures

We propose two extensions to the our approach. The first one concerns the duration of failures. Our framework tolerates one *permanent* processor failure. Relaxing this assumption to make the system tolerate one *transient* processor failure (one at a time of course) implies to address the following issue. After restarting the failed task on the spare processor, if the failure of the processor is transient, it could likely happen that the failed task restarts too, although probably in an incorrect state. Hence, a problem occurs when the former task updates its outputs since we would have *two tasks* updating the same output in parallel. This problem can be overcome by enforcing a property such that all tasks must check the shared variables FAILED and SPARE so that they can learn the status of the system and take a precaution if they have already been replaced by the monitor. When a task realizes that it has been restarted by the monitor, it must terminate immediately. In this case, since there is no more monitor in the system, the task terminates itself and restarts the monitor application, thus returning the system to its normal state where it can again tolerate one transient processor failure. The following code implements this action:

$$\begin{aligned} Rem_i \ = &\ \texttt{if}\ \text{FAILED} = i\ \texttt{and}\ \text{SPARE} \neq This\,Processor\ \ \texttt{then} \\ &\ \text{SPARE} := This\,Processor;\ \text{FAILED} := 0;\ \texttt{restart_monitor}\ ; \end{aligned}$$

where $This\,Processor$ denotes the ID of the processor executing that code and `restart_monitor` is a macro that terminates the task and restarts the monitoring application. The shared variable SPARE is initially set to the identification number of the spare processor. Assume that the task i has failed and has

been restarted on the spare processor. When the previous code is executed on the spare processor, it will see that even if FAILED is set to i, the task should not be stopped since it runs on the spare processor. On the other hand, the same task restarting after a transient failure on the faulty processor will detect that it must stop and will restart the monitor. The code Rem_i must be added to the program of τ_i before the output update:

$$\texttt{write}(o) \qquad \Longrightarrow \qquad Rem_i; \texttt{write}(o);$$

In order to detect any processor failure and to guarantee the real-time constraints, the duration of the transient failure must be larger than the max of the failure detection times α_i (*c.f.* equation 4 in Section V-A).

The second extension is to tolerate *several* failures at a time. We assumed that the system had one spare processor running a special monitoring program. In fact, additional spare processors could be added to tolerate more processor failures at a time. This does not incur any problem with our proposed approach. The only concern is the implementation of a coordination mechanism between the spare processors, in order to decide which one of them would resume the monitor application after the monitor processor has restarted a failed task τ_i.

VII. RELATED WORK

Related work on failure detectors is abundant. On the theoretical side, Fisher et al. have demonstrated that, in an asynchronous distributed systems (i.e., no global clock, no knowledge of the relative speeds of the processes or the speed of communication) with reliable communications (although messages may arrive in another order than they were sent),if one single process can fail permanently, then there is no algorithm which can guarantee consensus on a binary value in finite time [21]. Indeed, it is impossible to tell if a process has died or if it is just very slow in sending its message. If this delayed process's input is necessary, say, to break an even vote, then the algorithm may be delayed indefinitely. Hence no form of fault-tolerance can be implemented in totally asynchronous systems. Usually, one assumption is relaxed, for instance an upper bound on the communication time is known, and this is exactly what we do in this paper to design our failure detector. Then, Chandra and Toueg have formalized unreliable failure detectors in terms of completeness and accuracy [22]. In particular, they have shown what properties are required to reach consensus in the presence of crash failures. On the practical side, Aggarwal and Gupta present in [5] a short survey on failure detectors. They explain the push and pull methods in detail and introduces QoS techniques to enhance the performance of failure detectors.

Related work on automatic transformations for fault-tolerance include the work of Kulkarni and Arora [23]. Their technique involves synthesizing a fault-tolerant program starting from a fault-intolerant program. A program is a set of states, each state being a valuation of the program's variables, and a set of transitions. Two execution models are considered: high atomicity (the program can read and write any number of its variables in one atomic step, i.e., it can make a transition from any one state to any other state) and low atomicity (it can't). The initial fault-intolerant program ensures that its specification is satisfied in the absence of faults although no guarantees are provided in the presence of faults. A fault is a subset of the set of transitions. Three levels of fault-tolerance are studied: **failsafe ft** (in the presence of faults, the synthesized program guarantees safety), **non-masking ft** (in the presence of faults, the synthesized program recovers to states from where its safety and liveness are satisfied), and **masking ft** (in the presence of faults the synthesized program satisfies safety and recovers to states from where its safety and liveness are satisfied). Thus six algorithms are provided. In the high atomicity model (resp. low), the authors propose a sound algorithm that is polynomial (resp. exponential) in the state space of the initial fault-intolerant program. In the low atomicity model, the transformation problem is NP-complete. Each transformation involves recursively removing bad transitions. This principle of program transformation implies that the initial fault-intolerant program should be maximal (weakest invariant

and maximal non-determinism). Such an automatic program transformation is very similar to discrete controller synthesis.

Furthermore, Kulkarni and Ebnenasir study the automatic transformation of a fault-intolerant program (with the high atomicity execution model) into a **multi-tolerant** program [24]. This is a program that is failsafe tolerant to one class of faults, non-masking tolerant to another class of faults, and masking tolerant to still another class of faults. The technique is based on [23]. Finally, our program transformations are related to Software Thread Integration (STI) [25]. STI involves weaving a host secondary thread inside a real-time primary thread by filling the idle time of the primary thread with portions of the secondary thread. Compared to STI, our approach formalizes the program transformations and also guarantees that the real-time constraints of the secondary thread will be preserved by the obtained thread (and not only those of the primary thread).

VIII. Conclusion

In this paper, we presented a formal approach to fault-tolerance. Our fault-intolerant real-time application consists of periodic, independent tasks that are distributed onto processors showing omission/crash failure behavior, and of one spare processor for the hardware redundancy necessary to the fault-tolerance. We derived program transformations that automatically convert the programs such that the resulting system is capable of tolerating one permanent or transient processor failure at a time. Fault-tolerance is achieved by heartbeating and checkpointing/roll-back mechanisms. Heartbeats and checkpoints are thus inserted automatically, which yields the advantage of being transparent to the developer, and on a periodic basis, which yields the advantage of relatively simple verification of the real-time constraints. Moreover, we choose the heartbeating and checkpointing periods such that the overhead due to adding the fault-tolerance is minimized. We also proposed mechanisms to schedule all the detection tasks onto the spare processor, in such a way that the detection period is the same as the heartbeat period.

This transparent periodic implementation, however, has no knowledge about the semantics of the application and may yield large overheads. In the future, we propose to overcome this drawback by shifting checkpoint locations within a predefined safe time interval such that the overhead will be minimum. This work can also be extended to the case where processors execute multiple tasks with an appropriate scheduling mechanism. On the other hand, these fundamental fault-tolerance mechanisms can also be followed by other program transformations in order to tolerate different types of errors such as communication, data upsetting etc. These transformations, are seemingly more user dependent, which may led to the design of aspect-oriented based tools.

References

[1] F. Cristian, "Understanding fault-tolerant distributed systems," *Communication of the ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[2] V. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[3] P. Jalote, *Fault-Tolerance in Distributed Systems*. Prentice Hall, 1994.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[5] A. Aggarwal and D. Gupta, "Failure detectors for distributed systems," Tech. Rep., 2002, indian Institute of Technology, Kanpur, India.

[6] M. Aguilera, W. Chen, and S. Toueg, "Heartbeat: A timeout-free failure detector for quiescent reliable communication," in *Proceedings of the 11th International Workshop on Distributed Algorithms*. Springer-Verlag, 1997, pp. 126–140.

[7] A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Trans. Comput.*, vol. 46, no. 9, pp. 976–985, 1997.

[8] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489–510, Oct. 2000.

[9] M. Beck, J. Plank, and G. Kingsley, "Compiler-assisted checkpointing," University of Tennessee, Tech. Rep., 1994.

[10] L. Silva and J. Silva, "System-level versus user-defined checkpointing." in *Symposium on Reliable Distributed Systems*, 1998, pp. 68–74.

[11] H. Kopetz, *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Kluwer Acadamic Publishers, 1997.
[12] H. Nielson and F. Nielson, *Semantics with Applications—A Formal Introduction*. John Wiley & Sons, 1992.
[13] P. Puschner and A. Burns, "A review of worst-case execution-time analysis," *Real-Time Systems Journal*, vol. 18, no. 2-3, pp. 115–128, 1999.
[14] X. Li, T. Mitra, and A. Roychoudhury, "Modeling control speculation for timing analysis," *Real-Time Systems Journal*, vol. 29, no. 1, Jan. 2005.
[15] P. Puschner, "Transforming execution-time boundable code into temporally predictable code," in *Design and Analysis of Distributed Embedded Systems*, B. Kleinjohann, K. Kim, L. Kleinjohann, and A. Rettberg, Eds. Kluwer Academic Publishers, 2002, pp. 163–172, iFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).
[16] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. MIT Press, 1990.
[17] P. Caspi, C. Mazuet, R. Salem, and D. Weber, "Formal design of distributed control systems with lustre," in *International Conference on Computer Safety, Reliabilitiy, and Security, SAFECOMP'99*, ser. LNCS, no. 1698, Toulouse, France, Sept. 1999, pp. 396–409, crisys Esprit Project 25.514.
[18] J. Rushby, "Bus architectures for safety-critical embedded systems," in *International Workshop on Embedded Systems, EMSOFT'01*, ser. LNCS, vol. 2211. Tahoe City, USA: Springer-Verlag, Oct. 2001.
[19] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.
[20] G. C. Buttazzo, "Rate monotonic vs. edf: Judgment day." *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.
[21] M. Fisher, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.
[22] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.
[23] S. Kulkarni and A. Arora, "Automating the addition of fault-tolerance," in *6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'00*, ser. LNCS, M. Joseph, Ed., vol. 1926. Pune, India: Springer-Verlag, Sept. 2000, pp. 82–93.
[24] S. Kulkarni and A. Ebnenasir, "Automated synthesis of multitolerance," in *International Conference on Dependable Systems and Networks, DSN'04*. Firenze, Italy: IEEE, June 2004.
[25] A. G. Dean and J. P. Shen, "Hardware to software migration with real-time thread integration," in *EUROMICRO'98: Proceedings of the 24th Conference on EUROMICRO*. Washington, DC, USA: IEEE Computer Society, 1998, p. 10243.