Implementing Fault-Tolerance in Real-Time Systems by Automatic Program Transformations

Tolga Ayav
INRIA Rhône-Alpes
655 Avenue de l'Europe
38334 Saint-Ismier cedex, France
Tolga.Ayav@inrialpes.fr

Pascal Fradet
INRIA Rhône-Alpes
655 Avenue de l'Europe
38334 Saint-Ismier cedex, France
Pascal.Fradet@inrialpes.fr

Alain Girault
INRIA Rhône-Alpes
655 Avenue de l'Europe
38334 Saint-Ismier cedex, France
Alain.Girault@inrialpes.fr

ABSTRACT

We present a formal approach to implement and certify 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 establish correctness properties.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems, Real-time and embedded systems; D.4.5 [**Software**]: Operating Systems, Reliability [Checkpoint/restart, Fault-tolerance]; I.2.2 [**Computing Methodologies**]: Artificial Intelligence, Automatic Programming[Program transformation]

General Terms

Reliability, Languages.

Keywords

Fault-tolerance, Heartbeating, Checkpointing, Program transformations.

1. INTRODUCTION

In most distributed embedded systems, such as automotive and avionics, fault-tolerance is a crucial issue [9, 15]. It is defined as the ability of the system to comply with its specification despite the presence of faults in any of its components. 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 and the popular so-called "push" approach [1]. 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, and so on [16, 11]. The pros and cons of these strategies are discussed in [29].

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 failure of any other processor. Each transformation will implement a portion of either the detection or the masking of failures.

We consider only independent tasks, so 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. In Section 7, we report an experiment conducted with our approach, where tasks were *dependent*; for the sake of simplicity, we present in this paper our approach in the context of independent tasks.

The main benefit of our framework is the ability to formally prove that the transformed system satisfies some given real-time constraints even in the presence of one failure. The fault-tolerance techniques that we present (checkpointing, rollback, heartbeating, etc) are pretty standard in the OS

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

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. Other fault-tolerance techniques could have been considered as well.

Section 2 gives an overview of our approach. In Section 3, we give a formal definition for the real-time tasks and we introduce a simple programming language. Section 4 presents program transformations implementing checkpointing and heartbeating. We present the monitor task in Section 5, extend our approach to transient and multiple failures in Section 6, and outline a real case study implementation in Section 7. We finally review related work in Section 8 and conclude in Section 9.

2. OVERVIEW

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. We assume a reliable communication and a deterministic transmission time between the processors. These features can be realized, for instance, with FlexRay™ or CAN networks as will be mentioned in Section 6. For the sake of clarity, we assume a zero transmission time, but our results hold for non-zero transmission times as well.

Our failure assumption is that all the processors show omission/crash failure behavior [15]. 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 task model of [17]: all tasks are periodic and independent (*i.e.,* no precedence constraints). Periodic independent task set has been preferred in many works due to its simplicity in studying algorithms [22, 20]. We benefit from the same assumption to establish the theory, then propose several extensions to achieve a distributed system with dependent tasks. We present the CyCab application to demonstrate the implementation of our automatic transformation technique. More precisely, the program of each task has the form described in Figure 1. This programming model is adopted by control engineers and it is appropriate for real-time and reactive systems [14]. In particular, it has been succesfully deployed in the flight control system of the Airbus A340 and A380 planes [5].

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 or static scheduling approach) is still possible however.

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

Figure 1 Program model of periodic real-time tasks.

The task independency and single task per processor assumptions may seem too restrictive and unrealistic at first glance, but our technique can also be used with multiple dependent tasks as long as a static scheduling approach is exploited. Such a solution is discussed in Section 6. The implementation in CyCab mobile vehicle deals with three processors on which the communicating tasks are statically distributed, and demonstrates that our technique can be safely and efficiently used for enforcing basic fault-tolerance properties.

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 *worst case execution time* (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 one processor failure at a time. Masking of more than one transient processor failure at a time can be achieved with additional spare processors (see Section 6).

The stable memory is used to store the global state of each program. The global state provides masking of processor failures by rolling back to this global state as soon as a failure is detected. The stable memory also stores one shared variable per processor, used for failure detection. 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. This failure masking process is implemented by 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. 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 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.

3. TASKS

A real-time periodic task $\tau = (S, D)$ is specified by a program S and a period D. The program S is repeatedly executed each D 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 1). Each task also has a deadline $d \leq D$ 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, D) is that its program S must terminate before the end of its period D.

Programs are expressed in the following language:

S	::=	$x := A$	*assignment*
		`skip`	*no operation*
	|	`read`(i)	*input read*
	|	`write`(o)	*output write*
	|	S_1;S_2	*sequencing*
	|	`if` B `then` S_1 `else` S_2	*conditional*
	|	`for` $l = n_1$ `to` n_2 `do` S	*iteration*

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 [25] 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. Many real-time embedded systems have this structure: read an input i, perform some computation f, and return the result $f(i)$. Here, *Fac* should be seen as a generic computation simple enough to present concisely our techniques. Of course, as long as they are expressed in the previous syntax, much more complex and realistic computations could be treated as well.

```
Fac = read(i) ;
      if i > 10 then i := 10; o := 1; else o := 1;
      for  l = 1 to 10 do
           if l <= i  then o := o * l;
                      else skip;
      write(o);
```

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 WCET of any statement (or expression) S can be evaluated. The WCET analysis is the topic of much work (see [28, 21] 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:

$$\begin{array}{lcllcl}
\text{WCET}(x := e) & = & 3 & \text{WCET}(\texttt{skip}) & = & 1 \\
\text{WCET}(\texttt{read}) & = & 3 & \text{WCET}(\texttt{write}) & = & 3 \\
\text{WCET}(S_1;S_2) & = & \text{WCET}(S_1) + \text{WCET}(S_2) & & & \\
\text{WCET}(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2) & & & & & \\
\qquad & = & 1 + max(\text{WCET}(S_1), \text{WCET}(S_2)) & & & \\
\text{WCET}(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S) & & & & & \\
\qquad & = & (n_2 - n_1 + 1) \times (3 + \text{WCET}(S)) & & &
\end{array}$$

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 incrementing and testing the variable l in the for loop takes 3 time units.

With these figures, we get $\text{WCET}(Fac) = 83$. 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 $\{(S_1, D_1), \ldots, (S_n, D_n)\}$ is that each task must meet its deadline. Since each processor runs a single task, it amounts to:

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

The semantics of a statement S is given by the function $[\![S]\!] : \textbf{State} \rightarrow \textbf{State}$. A state $s \in \textbf{State}$ 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 [25]. The IO semantics of a task (S, D) 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. This property can be enforced by reading and storing the input value in the stable memory at the beginning of each period. 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 output of *Fac* is:

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

4. 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 a heartbeat and a checkpoint are executed every T_{HB} and T_{CP} period of time respectively. Conditional statements (`if`) complicate this insertion. They 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 statements. Such transformations suppose to be able to evaluate the WCET of programs. After this time equalization, checkpoints and heartbeats can be introduced simply using the same transformation. Another choice could have been to introduce heartbeats (resp. checkpoints) *at least* each T_{HB} (resp. T_{CP}). This choice does not require time equalization.

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

4.1 Equalizing execution time

Equalizing the execution time of a program consists in padding dummy code in less expensive branches. The dummy code added for padding is sequences of `skip` statements. We write $\texttt{skip}^n$ to represent a sequence of n `skip` statements: $\text{WCET}(\texttt{skip}^n) = n$. This technique is similar to the one used in "single path programming" [27].

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 [23]: cases are evaluated from top to bottom, and the transformation rule corresponding to the first pattern that matches the input program is performed.

$$
\begin{array}{lcl}
\mathcal{F}[\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2] & & \\
\qquad = \texttt{if } B \texttt{ then } \mathcal{F}[S_1]; \texttt{skip}^{\max(0,\delta_2-\delta_1)}; & & \\
\qquad\quad \texttt{else } \mathcal{F}[S_2]; \texttt{skip}^{\max(0,\delta_1-\delta_2)}; & & \\
\quad \text{with } \delta_i = \textsc{wcet}(\mathcal{F}[S_i]) \text{ for } i = 1,2 & & \\
\mathcal{F}[\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S] & = & \texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } \mathcal{F}[S] \\
\mathcal{F}[S_1;S_2] & = & \mathcal{F}[S_1];\ \mathcal{F}[S_2] \\
\mathcal{F}[S] & = & S \quad \text{otherwise}
\end{array}
$$

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.

It is easy to show that for any program S, the best (BCET) and worst (WCET) case execution times of $\mathcal{F}[S]$ are the same.

Property 1 $\forall S, \quad \textsc{bcet}(\mathcal{F}[S]) = \textsc{wcet}(\mathcal{F}[S])$.

In this case, the exact execution time (EXET) of a program is well-defined and equal to its BCET and WCET. Furthermore, the transformation $\mathcal{F}$ does not change the WCET of programs.

Property 2 $\forall S, \quad \textsc{wcet}(S) = \textsc{wcet}(\mathcal{F}[S])$.

Of course, the BCET of the transformed program will be greater than the BCET of the initial program, but this is the price to pay to guarantee a bound on the final fault-tolerant program, whatever the occurrences of the faults. We believe that, in the context of real-time and reactive embedded systems, this drawback is worth the benefit.

The interested reader will find the corresponding proofs (simple structural induction) in a companion paper [2]. The transformation applied on our example *Fac* produces:

```
Fac1 = F[Fac] read(i);
if i > 10 then i := 10; o := 1 else o := 1; skip^3;
for  l = 1 to 10 do
  if l <= i  then o := o * l else skip^3;
write(o);
```

4.2 Checkpointing and heartbeating

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

- `hbeat` sends a heartbeat telling the monitor that the processor is alive. This command is implemented by setting a special variable in the stable memory. The vector $\mathrm{HBT}[1 \ldots n]$ gathers the heartbeat variables of the n tasks. The command `hbeat` in task i is implemented as $\mathrm{HBT}[i] := 1$.

- `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 5 how to compute the optimal periods for those two commands, optimality being defined *w.r.t.* those real-time constraints.

We define below a transformation $\mathcal{I}_c^T(S,t)$ that inserts the command c every T units of time in the program S. 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 $\mathcal{I}$ will be used both for checkpointing and heartbeating. Again, the rules below must be understood like a case expression in ML:

1. $\mathcal{I}_c^T(S,t) \quad = S \quad$ if $\textsc{exet}(S) < t$

2. $\mathcal{I}_c^T(S,t) \quad = c;\mathcal{I}_c^T(S, T - \textsc{exet}(c) + t) \quad$ if $t \leq 0$

3. $\mathcal{I}_c^T(a,t) \quad = \quad a;c \quad$ if $0 < t \leq \textsc{exet}(a)$ and a is atomic

4. $\mathcal{I}_c^T(S_1;S_2,t) = \mathcal{I}_c^T(S_1,t);\mathcal{I}_c^T(S_2,t_1)$
with $t_1 = t - \textsc{exet}(S_1)$ if $\textsc{exet}(S_1) < t$ and
$t_1 = T - \textsc{exet}(c) - r$ if $\textsc{exet}(S_1) = t + q(T - \textsc{exet}(c)) + r$
with $q \geq 0$ and $0 \leq r < T - \textsc{exet}(c)$)

5. $\mathcal{I}_c^T(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, t)$
$\quad = \texttt{if } b \texttt{ then } \mathcal{I}_c^T(S_1, t-1) \texttt{ else } \mathcal{I}_c^T(S_2, t-1)$

6. $\mathcal{I}_c^T(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S, t)$
$\quad = \mathit{Fold}(\mathcal{I}_c^T(\mathit{Unfold}(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S), t))$

The transformation $\mathcal{I}$ relies on the property that all paths of the program have the same execution time and that the exact execution time (EXET) of any statement is well-defined (see Property 1 in Section 4.1). In order to insert heartbeats afterward, 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.

Rule 1 states that, when the statement S finishes before the next insertion time t (*i.e.,* $\textsc{exet}(S) < t$), the transformation terminates and nothing is inserted. In all the other cases (rules 2 to 6), the WCET of S is greater than t and at least one insertion must be performed.

Rule 2 applies when the time counter t is negative. This case may arise when the ideal point for inserting the command c is "in the middle" of the boolean expression of a conditional statement `if` . When t is negative, the command must be inserted right away. The transformation proceeds with the resulting program and the time target for the next insertion is reset to $T - \textsc{exet}(c) + t$, that is, it is computed *w.r.t.* the ideal previous insertion point to avoid any clock drift.

Rule 3 states that, when the program is an atomic command a (whose EXET is greater than or equal to t), the command c is inserted right after a, that is $(\textsc{exet}(a) - t)$ units of time later than the ideal point.

Rule 4 states that the insertion in a sequence $S_1;S_2$ is first done in S_1. The residual time t_1 used for the insertion in S_2 is either $(t - \textsc{exet}(S_1))$ if no insertion has been

performed inside S_1 or $(T - \text{EXET}(c) - r)$ if r is the time residual remaining after the $q+1$ insertions inside S_1 (*i.e.,* if $\text{EXET}(S_1) = t + q(T - \text{EXET}(c)) + r$).

Rule 5 states that, for conditional statements, the insertion is performed in both branches. The time of the test and branching is taken into account by decrementing the time residual $(t-1)$.

Rule 6 applies to loop statements. It unrolls the loop completely (thanks to the *Unfold* operator), performs the insertion in the unrolled resulting program and then factorizes code by folding code in `for` loops as much as possible (thanks to the *Fold* operator). The *Unfold* and *Fold* operators are defined by the following transformation rules:

$$\begin{array}{l} \mathit{Unfold}(\texttt{for } l = n_1 \texttt{ to } n_2 \texttt{ do } S) = \\ \qquad l := n_1;\ S;\ l := n_1 + 1;\ S;\ \ldots\ l := n_2;\ S, \\ \mathit{Fold}((\texttt{for } l = 1 \texttt{ to } n \texttt{ do } S);\ l := n+1; S) = \\ \qquad\qquad\qquad \texttt{for } l = 1 \texttt{ to } n+1 \texttt{ do } S. \end{array}$$

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. Other solutions preventing the possibility of code explosion exist (*e.g.,* padding the body of loops to perform the insertion at a fixed place).

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

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:*

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

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

The formalization and proof of Property 3 is beyond the scope of this paper and can be found in [2].

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:

$$T'_{CP} = \frac{T_{CP}}{1 + \frac{\overline{h}}{T_{HB}-\overline{h}}} \tag{2}$$

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 D. Hence, k has to be computed as:

$$k = \left\lceil \frac{D - \text{WCET}(S''; \texttt{hbeat})}{T_{HB}} \right\rceil \tag{3}$$

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: $T_{CP} \leq \Delta_{CP} < T_{CP} + \varepsilon + \overline{h}$ and $T_{HB} \leq \Delta_{HB} < T_{HB} + \varepsilon$.*

This property is a corollary of property 3 (see [2] for the proof).

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:

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

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

$$\begin{array}{ll} \bar{h} = \text{WCET}(\texttt{hbeat}) = 3, & \bar{c} = \text{WCET}(\texttt{checkpt}) = 10, \\ T_{HB} = 10, & T_{CP} = 80. \end{array}$$

So, we get $T'_{CP} = 80 - \frac{3 * T'_{CP}}{10-3} = 56$ hence $\mathcal{I}^{56}_{\texttt{checkpt}}(\mathit{Fac}_1, 56)$ produces:

$$\begin{array}{l} \mathit{Fac}_2\ = \texttt{read}(i); \\ \texttt{if } i > 10 \texttt{ then } i := 10;\ o := 1 \texttt{ else } o := 1;\ \texttt{skip}^3; \\ \texttt{for } \ l = 1 \texttt{ to } 6 \texttt{ do} \\ \quad \texttt{if } l <= i \ \texttt{ then } o := o * l \texttt{ else skip}^3; \\ l := 7;\ \texttt{if } l <= i \ \texttt{ then checkpt};\ o := o * l; \\ \qquad\qquad \texttt{else checkpt};\ \texttt{skip}^3; \\ \texttt{for } \ l = 8 \texttt{ to } 10 \texttt{ do} \\ \quad \texttt{if } l <= i \ \texttt{ then } o := o * l \texttt{ else skip}^3; \\ \texttt{write}(o); \end{array}$$

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

```
Fac3  = hbeat; read(i);
 if i > 10 then i := 10; hbeat; o := 1;
          else o := 1; hbeat; skip^3;
 for  l = 1 to 6 do
      if i > 0  then hbeat; o := o * l;
                else hbeat; skip^3;
 l := 7; if i > 0  then hbeat; checkpt_1; hbeat;
                        checkpt_2; o := o * l;
                   else hbeat; checkpt_1; hbeat;
                        checkpt_2; skip^3;
 for  l = 8 to 10 do
      hbeat;
      if i > 0  then o := o * l; else skip^3;
 write(o); hbeat; skip^5; hbeat;
```

In Fac_3, the checkpoint is performed after 83 units of time in both branches, which is inside the $[80, 86)$ interval of Property 4. Finally, since $\text{WCET}(Fac_3) = 143$ and the period is 200, Equation (3) gives $\lceil \frac{200-143}{10} \rceil = 6$, so the last `hbeat` must be changed into `hbeat`(6). Figure 2 illustrates the form of a general program (*i.e.,* not Fac_3) after all the transformations:

5. IMPLEMENTING THE MONITOR

The monitor, executed on the spare processor, performs failure detection by checking the heartbeats and performs a roll-back recovery in case of a failure. In the following subsections, we explain heartbeat detection and roll-back recovery actions, together with the implementation details and conditions for real-time guarantee.

5.1 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:

$$\forall i \in \{1, 2, ..., n\}, \text{WCET}(Det_i) \leq T_{HB_i}. \quad (4)$$

Preemptive scheduling techniques such as Rate-Monotonic (RM) and Earliest-Deadline-First (EDF) settle the problem. In our context, RM guarantees that Γ is schedulable if:

$$U = \sum_{i=1}^{n} \frac{\text{WCET}(Det_i)}{T_{HB_i}} \leq 2(2^{1/n} - 1) \quad (5)$$

Under the same assumptions, EDF guarantees that Γ is schedulable if $U \leq 1$. These schedulability conditions highlight that EDF allows a better processor utilization while both are appropriate and sufficient for scheduling the monitoring tasks with deadline guarantee.

The program Det_i is:

```
Det_i  = HBT[i] := HBT[i] − 1;
         if HBT[i] = −2 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:

$$\alpha_i < 3(T_{HB_i} + \varepsilon) + L_r + L_w + 3\xi_i \quad (6)$$

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* [7], 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). 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).

5.2 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 (x) = FAILED:=x; `restart` (τ_x, 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

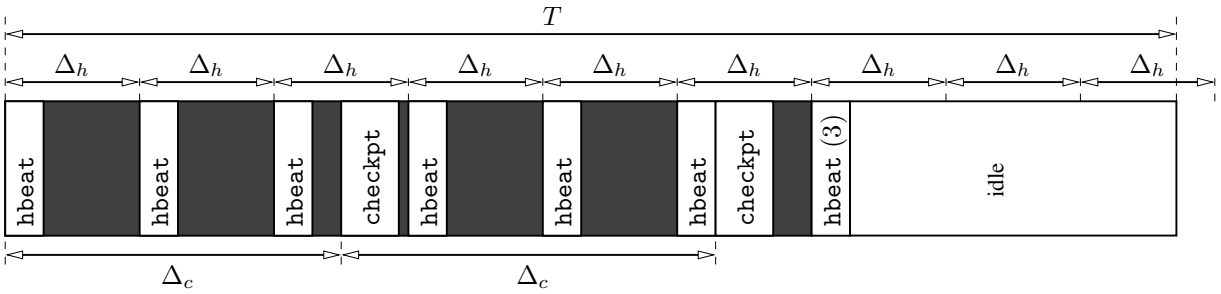

Figure 2: Program with checkpointing and heartbeating.

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:

$$\beta = 3(T_{HB} + \varepsilon) + T_{CP} + L_r + L_w + L_C + 3 \max_{1 \leq i \leq n} \xi_i + \text{WCET}(Det) + \text{WCET}(Rec) \quad (7)$$

5.3 Satisfying the real-time constraints

After the program transformations, the WCET of the fault-tolerant program of the task (S'', D), taking into account the recovery time, is given by the following expression:

$$\bar{S}'' = \bar{S} + \left\lfloor \frac{\bar{S}}{T_{CP}} \right\rfloor \times \bar{c} + \left(\frac{\bar{S}}{T_{HB}} + 1 \right) \times \bar{h} + \beta \quad (8)$$

where $\bar{S}$ and $\bar{S}''$ denote the WCETs of S and S'' respectively. 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 Equations (8) and (7), we obtain a two-value function $f(T_{CP}, T_{HB})$ of the form:

$$f = \frac{\bar{S} \times \bar{c}}{T_{CP}} + \frac{\bar{S} \times \bar{h}}{T_{HB}} + 3T_{HB} + T_{CP} + K \quad (9)$$

where K is a constant. Note that neglecting the floor function in Equation (8) is for the purpose of an explicit calculation and causes approximate results. Figure 3 below illustrates the f function.

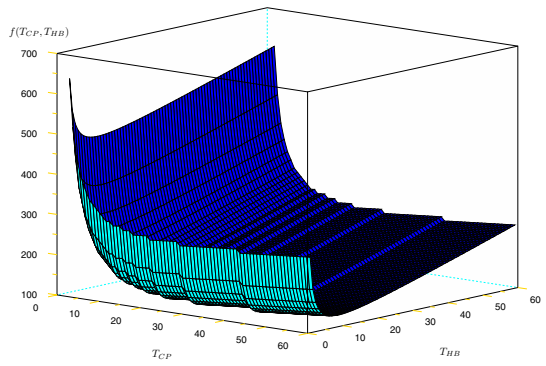

Figure 3: $f(T_{CP}, T_{HB})$.

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:

$$\frac{\partial f}{\partial T_{CP}} = 1 - \frac{\bar{S} \times \bar{c}}{T^2_{CP}}, \quad \frac{\partial f}{\partial T_{HB}} = 3 - \frac{\bar{S} \times \bar{h}}{T^2_{HB}} \quad (10)$$

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 *i.e.,*:

$$\left.\frac{\partial f}{\partial T_{CP}}\right|_{T_{CP}=T^{\star}_{CP}} = 0 \text{ and } \left.\frac{\partial f}{\partial T_{HB}}\right|_{T_{HB}=T^{\star}_{HB}} = 0$$

Hence, Equations (11) and (12) give the optimal values for the heartbeat and checkpoint periods:

$$T^{\star}_{CP} = \sqrt{\bar{S} \times \bar{c}} \quad (11)$$

$$T^{\star}_{HB} = \sqrt{\frac{1}{3}\bar{S} \times \bar{h}} \quad (12)$$

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.

Similar analysis and results are described in [26]. However, they do not have heartbeats and only consider the optimal placement of checkpoints. To the best of our knowledge, our result is the first to provide the optimal period for both the heartbeating and the checkpointing in order to minimize the WCET of the transformed program, whatever the occurrences of the faults (*i.e.,* taking into account the detection and recovery delays).

Finally, in order to satisfy the real-time property of the whole system, the only criterion that should be checked is:

$$f(T_{CP_i}, T_{HB_i}) < D_i, \qquad \forall i \in \{1, 2, \ldots, n\} \quad (13)$$

Removing the assumption of zero communication time just involves adding a worst case communication delay parameter in Equations (6) and (7), which has no effect on the optimum values $T^{\star}_{CP}$ and $T^{\star}_{HB}$.

6. EXTENSIONS

6.1 Tolerating transient failures

We have studied two main extensions of our technique and have applied it to a real case study. The first extension 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:

$Rem_i =$ `if` FAILED $= i$ `and` SPARE $\neq$ *This Processor* `then`
SPARE := *This Processor*;
FAILED := 0;
`restart_monitor` ;

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:

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

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 (6) in Section 5.1).

6.2 Tolerating several failures at a time

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 can be added to tolerate more processor failures at a time. This does not incur any problem with our proposed approach. It only requires the implementation of a coordination mechanism between the spare processors, in order to decide which one of them should resume the monitor application after the monitor processor has restarted a failed task τ_i. This issue can be easily solved by statically ordering the processors.

7. APPLICATION: THE CYCAB VEHICLE

We give a brief discussion about the implementation of the proposed methods through a real application called CyCab. Additional details are available in [2]. The CyCab is a small vehicle designed for transportation in downtown areas, amusement parks *etc.* [4]. The physical architecture consists of two MPC555 micro-controllers, a PC, and a CAN bus. We call these nodes `F555`, `R555`, and `ROOT` respectively.

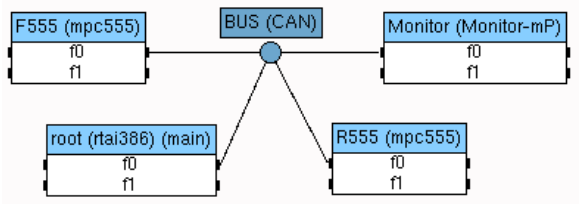

Figure 4: Architecture graph of the CyCab application.

In order to implement our program transformations, we have added one more node, named `MONITOR` (see Figure 4).

We have used as a case study the "manual-driving" application. It is specified as a data-dependency graph, whose nodes are atomic computation tasks (written in C) and whose edges are data-dependencies between these tasks. The WCET of these atomic tasks onto the nodes range between $0.2ms$ and $0.6ms$. Also, because some tasks use some specific sensors or actuators that are physically located on the CyCab vehicle, they can only be executed on some of the nodes; in that case, their WCET is set to ∞ on all the other nodes. Finally, we have chosen, for this case study, to work at the task level, *i.e.*, not looking at the code *inside* the tasks. The benefit of working at this coarse level is that the computational cost of applying our program transformations will be low. The drawback is that the ε parameter (equal to the min of the WCETs of all the tasks, *i.e.*, $0.2ms$) will be rather big, therefore making it difficult to be optimal since we will not be able to insert the heartbeats and checkpoints exactly at the designated points.

We start by distributing this application on the architecture of Figure 4 using the SynDEx tool that supports the Algorithm Architecture Adequation (AAA) methodology [13]. AAA takes the application graph and distributes it onto the given architecture. It is based on graph models to exhibit the potential parallelism of the application, and the implementation is formalized in terms of graph transformations. In our case, we obtain a distributed static schedule of the tasks on the architecture, with a global WCET of respectively $4.19ms$, $3.05ms$, and $4.10ms$ on `F555`, `R555`, and `ROOT`.

The advantage of applying our program transformations *after* the distribution of the application by SynDEx is that each node (`F555`, `R555`, and `ROOT`) now has a single static schedule that follows the basic model of Figure 1. So we can apply our program transformations to insert heartbeating and checkpointing. Note that, since the data-dependency graph of the application has neither conditional (`if`) nor iteration (`for`) edges, it is not necessary to apply our padding code transformation. However, since the application consists of dependent tasks, our technique needs to be slightly modified as follows. First, we fill the idle times between tasks with no-operations. Then, task dependency may cause new idle times after placing a checkpoint or heartbeat, since an insertion slightly changes the static schedule. So, after each insertion, the resulting static schedule is checked once more and all idle times are filled again before continuing with the next insertion.

The WCET of heartbeating is equal to $0.06ms$ on each node. The WCET of checkpointing is also equal to $0.06ms$ on each node. Furthermore, the worst-case communication time between any heartbeat task and its corresponding mon-

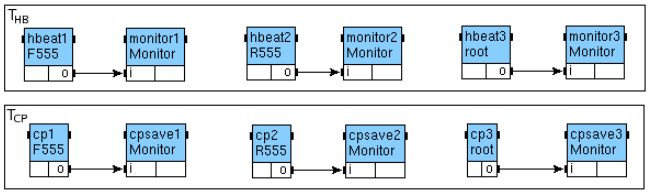

Figure 5: Application graphs for heartbeating and checkpointing.

itor task is equal to $0.12ms$. The worst-case communication time between any checkpoint task and its corresponding stable memory save task is equal to $0.16ms$. Here, we have chosen to use SynDEx to schedule the monitor tasks instead of a Rate Monotonic policy. This is achieved by adding the data-dependency graph of Figure 5. With these figures, $\bar{c}$ and $\bar{h}$ are respectively equal to $0.18ms$ and $0.21ms$. We thus compute $T^*_{CP} = 1.54ms$ and $T^*_{HB} = 1.42ms$.

With these figures, applying our automatic transformations to the schedule of each node produces a new static distributed schedule whose WCET is equal to $6.21ms$. The overhead due to the fault-tolerance is therefore $6.21 - 4.10 = 2.11ms$.

8. 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 [12]. 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, 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 [8]. 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 [1] 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.

Other works on failure recovery include the efforts of reserving sufficient slack in dynamic schedule, *i.e.,* gaps between tasks due to the precedence, resources or timing constraints, so that the scheduler can re-execute faulty tasks without jeopardizing the deadline guarantees [24]. Further studies proposed different heuristics for re-execution of faulty tasks in imprecise computation models such that faulty mandatory sub-tasks may supersede optional sub-tasks [3]. In contrast, our work is entirely in the static scheduling context.

Bronevetsky et al. have designed and implemented a precompiler for C/MPI programs in order to automatically insert checkpoints [6]. The target architecture consists of several fail-silent processors connected by a reliable message delivery system, and equipped with a distributed failure detector mechanism for detecting failed processes. Since the tasks can exchange messages, the local checkpoints must be coordinated to form a consistent global checkpoint. To achieve this, the authors have developed a new protocol for non-blocking coordination that works smoothly with application-level checkpointing. Yet, checkpoint insertion is not fully automatic since the programmer must manually insert calls to a function called `PotentialCheckpoint` at points in the application where he/she wants checkpointing to occur. Also, the authors do not consider the issue of real-time constraints.

Regarding automatic transformations for fault-tolerance, our work is related to the work of Kulkarni and Arora [18]. 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.

Furthermore, Kulkarni and Ebnenasir study the automatic transformation of a fault-intolerant program (with the high atomicity execution model) into a *multi-tolerant* program [19]. 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 [18]. Finally, our program transformations are related to Software Thread Integration (STI) [10]. 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).

9. CONCLUSION

We have presented a formal approach to implement classical fault-tolerance techniques in real-time systems. Our fault-intolerant real-time application is distributed onto processors showing omission/crash failure behavior, and of one spare processor for the hardware redundancy necessary to the fault-tolerance. We have 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. For this purpose, heartbeats and checkpoints are 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.

Choosing the lengths of checkpointing and heartbeating intervals is delicate. Long intervals lead to long roll-back time, while too frequent checkpointing leads to high overheads. We derived formulae for choosing the optimal checkpointing and hearbeating intervals. As a result, the overhead due to adding the fault-tolerance is minimized.

To the best of our knowledge, the two contributions presented in this article (*i.e.,* the formalization of adding fault-tolerance with automatic program transformations, and the computation of the optimal checkpointing and heartbeating periods to minimize the fault-tolerance overhead) are novel.

We have 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.

Finally, we have shown with a case-study that our work can be extended to the case where processors execute multiple tasks with an appropriate static scheduling mechanism.

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 further reduced.

Finally, these fundamental fault-tolerance mechanisms can be followed by other program transformations in order to tolerate different types of faults such as communication, data upsetting etc. These transformations are seemingly more user dependent, which may lead to the design of aspect-oriented based tools.

10. REFERENCES

[1] Aggarwal, A., and Gupta, D. Failure detectors for distributed systems. Tech. rep., Indian Institute of Technology, Kanpur, India, 2002. http://resolute.ucsd.edu/diwaker/publications/ds.pdf.

[2] Ayav, T., Fradet, P., and Girault, A. Implementing fault-tolerance in real-time systems by program transformations. Research Report 5743, Inria, May 2006. http://hal.inria.fr/inria-00077156/en.

[3] Aydin, H., Melhem, R., and Mossé, D. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Real-Time Computing Systems and Applications, RTCSA'00* (Cheju Island, South Korea, 2000), IEEE, pp. 289–296.

[4] Baille, G., Garnier, P., Mathieu, H., and Pissard-Gibollet, R. Le Cycab de l'Inria Rhône-Alpes. Technical report 0229, Inria, Rocquencourt, France, Apr. 1999. http://hal.inria.fr/inria-00071193/en.

[5] Brière, D., Ribot, D., Pilaud, D., and Camus, J.-L. Methods and specifications tools for Airbus on-board systems. In *Avionics Conference and Exhibition* (London, UK, Dec. 1994), ERA Technology.

[6] Bronevetsky, G., Marques, D., Pingali, K., and Stodghill, P. Automated application-level checkpointing of MPI programs. In *Principles and Practice of Parallel Programming, PPoPP'03* (San Diego, USA, June 2003), ACM, pp. 84–94.

[7] Caspi, P., Mazuet, C., Salem, R., and Weber, D. Formal design of distributed control systems with Lustre. In *International Conference on Computer Safety, Reliabilitiy, and Security, SAFECOMP'99* (Toulouse, France, Sept. 1999), no. 1698 in LNCS, pp. 396–409.

[8] Chandra, T., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *J. of the ACM 43*, 2 (Mar. 1996), 225–267.

[9] Cristian, F. Understanding fault-tolerant distributed systems. *Comm. of the ACM 34*, 2 (Feb. 1991), 56–78.

[10] Dean, A., and Shen, J. Hardware to software migration with real-time thread integration. In *Proc. of the 24th Conf. on EUROMICRO* (1998), IEEE Computer Society, p. 10243.

[11] Elnozahy, E., Alvisi, L., Wang, Y., and Johnson, D. A survey of rollback recovery protocols in message passing systems. *ACM Comp. Survey 34*, 3 (Sept. 2002), 375–408.

[12] Fisher, M., Lynch, N., and Paterson, M. Impossibility of distributed consensus with one faulty process. *J. of the ACM 32*, 2 (1985), 374–382.

[13] Grandpierre, T., Lavarenne, C., and Sorel, Y. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Int. Workshop on Hardware/Software Co-Design, CODES'99* (Rome, Italy, May 1999), IEEE.

[14] Halbwachs, N. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *International Conference on Computer-Aided Verification, CAV'98* (Vancouver, Canada, June 1998), vol. 1427 of *LNCS*, Springer-Verlag.

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

[16] Kalaiselvi, S., and Rajaraman, V. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana 25*, 5 (Oct. 2000), 489–510.

[17] Kopetz, H. *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[18] Kulkarni, S., and Arora, A. Automating the addition of fault-tolerance. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'00* (Pune, India, Sept. 2000), M. Joseph, Ed., vol. 1926 of *LNCS*, Springer-Verlag, pp. 82–93.

[19] Kulkarni, S., and Ebnenasir, A. Automated synthesis of multitolerance. In *Int. Conf. on Dependable Systems and Networks, DSN'04* (Firenze, Italy, June 2004), IEEE.

[20] Leistman, A., and Campbell, R. A fault-tolerant scheduling problem. *IEEE Trans. on Software Engineering 12*, 11 (1986), 1088–1089.

[21] Li, X., Mitra, T., and Roychoudhury, A. Modeling control speculation for timing analysis. *Real-Time Systems Journal 29*, 1 (Jan. 2005).

[22] Maheswaran, M., Ali, S., Siegel, H., Hensgen, D., and Freund, R. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop* (1999), pp. 30–44.

[23] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, 1990.

[24] Mossé, D., Melhem, R., and Ghosh, S. A nonpreemptive real-time scheduler with recovery from transient faults and its implementation. *IEEE Trans. on Software Engineering 29*, 8 (2003), 752–767.

[25] Nielson, H., and Nielson, F. *Semantics with Applications—A Formal Introduction*. John Wiley & Sons, 1992.

[26] Punnekkat, S., and Burns, A. Analysis of checkpointing for schedulability of real-time systems. In *Proc. of the Int. Workshop on Real-Time Computing Systems and Applications, RTCSA'97* (1997), pp. 198–205.

[27] Puschner, P. Transforming execution-time boundable code into temporally predictable code. In *Design and Analysis of Distributed Embedded Systems*. Kluwer, 2002, pp. 163–172.

[28] Puschner, P., and Burns, A. A review of worst-case execution-time analysis. *Real-Time Systems Journal 18*, 2-3 (1999), 115–128.

[29] Silva, L., and Silva, J. System-level versus user-defined checkpointing. In *Symp. on Reliable Distributed Systems, SRDS'98* (West Lafayette (IN), USA, Oct. 1998), pp. 68–74.