Static fault-tolerant scheduling with "pseudo-topological" orderings

Cătălin Dima[1], Alain Girault[2], and Yves Sorel[5]

[1] Université Paris 12, 61 av du Général de Gaulle, 94010 Créteil cedex, France
[2] INRIA Rhne-Alpes, 655 Av. de l'Europe, 38334 Saint-Ismier cedex, France
[3] INRIA Rocquencourt, B.P.105 - 78153, Le Chesnay cedex, France

Abstract. We give a graph-theoretical model for off-line fault-tolerant scheduling of dataflow algorithms onto multiprocessor architectures with distributed memory. Our framework allows the modeling of both processor and channel failures of the "fail silent" type (either transient or permanent), and failure masking is achieved by replicating operations and data communications. We show that, in general, the graph representing a fault-tolerant scheduling may have circuits; hence, the classical computation of starting/ending times, based upon a topological ordering, is inapplicable. We then provide a notion of "pseudo-topological ordering" that permits the computation of the starting/ending times even in the case of cyclic graphs. We also derive algorithms for computing the timeouts that are used for failure detection.

1 Introduction

Embedded systems are systems built for controlling physical hardware in a changing environment, and therefore are required to react to environmental stimuli within limited time. Therefore, the design of such systems requires thorough temporal analysis of the expected reactions. On the other hand, limitations on hardware reliability may require additional fault-tolerance design techniques and need of distributed architectures.

The complexity of the design of such a system is usually coped with by decomposing the problem into subproblems. Usually, the fault-tolerance problem is the hardest to address: one solution is to handle it at the level of hardware; see [15, 14, 20], to cite only a few. This solution has been extensively studied in the past decade and is very useful for ensuring reliable data transmission, fail-signal, or fail-silent "processor blocks", fault masking with "Byzantine" components and membership protocols. This approach is important as it hides a wealth of mechanisms for fault-tolerance from the software design; however, the types of failures it cannot handle need to be coped with at the design level.

Another solution is to combine scheduling strategies with fault detection mechanisms [10, 11, 17, 16]. Such a solution is suited to system architectures in which communications are supposed to be reliable and communication costs are small. Moreover, dynamic schedulers require the existence of a reliable "master" component. All these induce some limitations on the use of such techniques in embedded systems.

A third solution is to introduce redundancy levels at the design time [19, 12, 2]. Fault masking strategies like primary/backup or triple modular redundancy (multiple if more than one failure has to be tolerated), voting components, or membership protocols are then designed for failure masking. The scheduling strategies take into account durations of communications, and therefore this approach seems to be the most appropriate to embedded systems in which limited computational power of part of the processors is combined with tight real-time constraints at the response level. The drawback of this solution is that it does not take into account channel failures.

Taking into account both processor and channel failures is crucial in some highly critical and/or non-maintainable embedded systems, like vehicle control systems. Usually, this is achieved at the architecture level by triplicating the number of buses. But sometimes even triplication might not assure enough reliability, or just a more involved architecture is imposed, and then fail-silent channels must be taken into account during scheduling. On the other hand, in complex architectures, data transmission may involve routings, therefore specific fault-tolerant routing techniques [22, 4] may need to be applied. This problem is related to the so-called edge k-connectivity problem for graphs [3].

We propose a theoretical model for the problem of fault-tolerant static scheduling onto a distributed architecture. The basic idea is that a fault-tolerant scheduling will be the *union of several non-fault tolerant schedulings*, one for each possible failure patterns. Our model uses only basic notions from graph theory and a calculus with mins and maxs for computing execution times. We consider distributed architectures in which the memory is distributed too, in which we only abstract from the communication protocols onto each channel and the execution schemes on each "processor node". Our model does not include any centralised control for detecting failures, or "membership protocol" for exchanging information about failure occurrence. On the other hand, we abstract from the "local fault-tolerance" mechanisms: we work with fail-silent components, an assumption which hides the mechanisms for achieving such behaviour[4]. But this is fine, since our aim is not to provide a model for any fault-tolerant problem, but mainly for those in which fault-tolerance must be achieved for the whole system, hence requiring a global approach.

We use timeouts in order to detect failures of data communications, but also in order to propagate failure information. The reason is that, in our replication technique, the replicas of the same data transmission do not necessarily have the same source, the same destination, or the same route. Therefore, some replicas of the same task may "starve" because their needed data do not arrive, and therefore they will not be able to send their data as well, propagating further this starvation to other tasks. This propagation is achieved with timeouts.

The theoretical problem raised by this approach is that, due to the need of redundant routings, we may have to schedule non-acyclic graphs. In classical scheduling theory, cyclic graphs pose the problem of the *existence* of a scheduling.

[4] Recent studies on modern processors have shown that fail-silence can be acheived at a reasonable cost [1].

Thanks to the construction principles of our fault-tolerant schedulings, this will not be the case in our framework, and this forms the core of our work and hence one of our original contributions.

We solve this problem by introducing an original notion of *pseudo-topological ordering*, which models the "state-machine" arbitration mechanism between replicas of the same operation. We show that each scheduling with replication can be ordered pseudo-topologically, and therefore there exists a minimal execution of it. We also prove the existence of the minimal timeout mapping, which associates to each scheduled operation 1+ the latest time when it should start its execution, in any of the failure patterns that must be tolerated (all the execution times are expressed in time units, i.e., integers, hence the "1+"). These results are based upon two algorithms that can be seen as a basic technique for constructing a fault-tolerant scheduling.

Though we have named it a "static scheduling" model, the behaviour of each node is quite dynamic as we construct "decision makers" on each component, capable of choosing at run time the sequence of operations to be done on that component. Moreover, both processor and channel failures are treated uniformly in our model, it is only at the implementation that they will differentiate. In this sense, we also comment on the restrictions that this uniform treatment imposes on the implementations.

Finally, let us mention that this paper gives the theoretical framework for the heuristics studied in a previously published paper [5].

The paper is organised as follows: in the second section we define the problem of non-fault-tolerant scheduling of a set of tasks with precedence relations. The third section contains our graph-theoretic model of fault-tolerant scheduling, together with its principles of execution, formalised in the form of a min-max calculus of starting and ending times. The fourth section deals with the formalisation of the failure detection and propagation mechanisms. The core of this section is the computation of the *minimal timeout mapping*, used for failure detection. We end with some conclusions and future work.

2 Plain schedulings

This section is a brief review of some well-known results from scheduling theory, presented in a perhaps different form in order to be further generalised to fault-tolerant scheduling. For reasons of convenience in defining and manipulating schedulings, and especially due to the possibility of having data dependencies routed through several channels, we chose to represent programs as bipartite *dags* (directed acyclic graphs):

Definition 1. *A* ***task dag*** *is a bipartite dag* $G_A = (O_A, D_A, E_A)$.

Nodes in O_A are tasks and nodes in D_A are data dependencies between tasks; e.g., $(t_1, d_1), (d_1, t_2) \in E_A$, with $t_1, t_2 \in O_A$ and $d_1 \in D_A$ means that the task t_1, at the end of its execution, sends data to the task t_2, which waits for the reception of this data before starting its execution.

We will rely also on a notion of *architecture graph*, which is simply a bipartite undirected graph. Figure 1(left) represents an example of task dag. Here, $O_A = \{A, B, C, D\}$ and $D_A = \{A{\to}B, A{\to}C, B{\to}D, C{\to}D\}$. Figure 1(right) represents an example of an architecture graph. Here, P_1, P_2, P_3 and P_4 are the processors and C_1 and C_2 are the communication links.

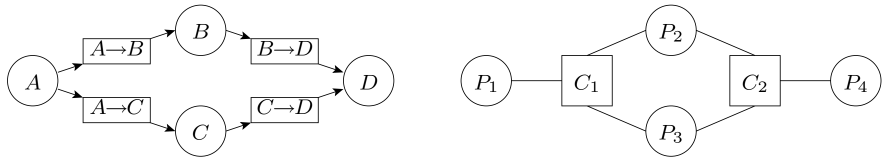


Fig. 1. Left: a task dag; Right: an architecture graph.

In the field of data-flow scheduling, each operation is placed onto some processor and each data dependency onto some channel, hence yielding a kind of copy of the task dag G_A, in which it is possible to have some data dependencies routed through several channels between their source and destination. Here we will call such a "placement" a *plain scheduling* and define it as follows:

Definition 2. *A **plain scheduling** of a task dag $A = (O_A, D_A, E_A)$ is a labelled dag $G = (V, E, \lambda)$ with $\lambda : V \longrightarrow O_A \cup D_A$, satisfying the following properties:*

- S1 *For each $v \in V$ such that $\lambda(v) = a \in O_A$, for each b such that $(b, a) \in E_A$, there exists $v' \in V$ with $\lambda(v') = b$ and $(v', v) \in E$. That is, each scheduled operation must receive all its incoming data dependencies.*
- S2 *For each $v \in V$ such that $\lambda(v) = a \in D_A$, for each b such that $(b, a) \in E_A$, there exists a sequence $v_1, \ldots, v_k \in V$ with $v_k = v$, $\lambda(v_1) = b$, $\lambda(v_2) = \ldots = \lambda(v_k) = a$ and $(v_i, v_{i+1}) \in E$ for all $1 \le i \le k-1$. That is, data transmissions may be routed through several channels from their source to their destination.*
- S3 *For each $a \in O_A$ there exists exactly one $v \in V$ such that $\lambda(v) = a$. That is, each operation is scheduled exactly once.*
- S4 *For each $a \in D_A$, the set $\{v \in V \mid \lambda(v) = a\}$ forms a linear subgraph in G (and is nonempty by condition S2). That is, the routings of any given data dependency go directly from the source operation to the destination operation.*

*Given an architecture graph $G_S = (P, C, E_S)$ and a mapping $\pi : V \longrightarrow P \cup C$, we say that the pair (G, π) is a **plain scheduling onto** G_S if π satisfies the following condition:*

$$\forall v \in V, \lambda(v) \in O_A \Longrightarrow \pi(v) \in P$$

Several clarifications and comments are needed concerning this definition. Firstly, note that in our definition, the architecture is "hidden" within the edges of the graph A. Moreover, the *scheduling order* on each processor is abstracted as edges (v, v') for which $(\lambda(v), \lambda(v')) \notin E$ and $\lambda(v) \neq \lambda(v')$.

Then observe that properties S1 and S2 show the difference between how operations and data dependencies are scheduled, hence the "bipartite dag" model for task dags. Finally, note that there can be nodes $v \in V$ such that $\lambda(v) \in D_A$ (that is, carrying a data dependency) and $\pi(v) \in P$ (that is, scheduled onto some processor). These nodes model two things: reroutings between adjacent channels, and intra-processor transmissions of data.

Figure 2 represents *one* plain scheduling of the task dag of Figure 1(left) onto the architecture graph of Figure 1(right). The notation "A/P_4" means that $\pi(A) = P_4$; ovals are operations scheduled onto some processor (e.g., A/P_4); square rectangles are data dependencies scheduled onto some communication channel (e.g., $A{\rightarrow}B/C_1$); and rounded rectangles are data dependencies scheduled onto some processor for re-routing (e.g., $A{\rightarrow}B/P_3$). A plain scheduling, in our sense, is the *result* of some static scheduling algorithm. The restrictions on the placement of tasks or data dependencies, which guide the respective algorithm, are not visible here – we only have the result produced by such restrictions.

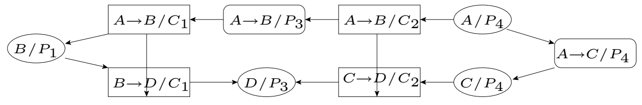


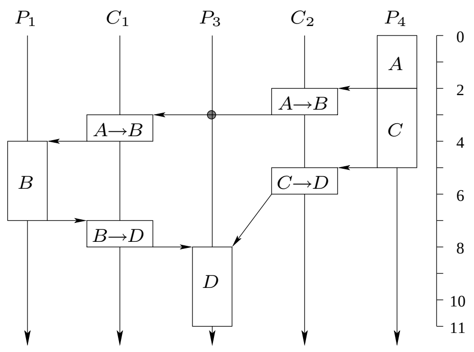
Fig. 2. A plain scheduling of the task dag onto the architecture graph of Figure 1.

Once a plain scheduling is created and we know the execution time of each task and the duration of sending each data dependency onto its respective channel, we may compute the starting time for each of them by a least fix-point computation, based upon the following definition of an execution:

An **execution** of a plain scheduling onto a distributed architecture is governed by the following three principles:

P1 Each task is executed only after receiving all the data dependencies it needs.
P2 Each data dependency is executed only after the task issuing it terminates.
P3 Each operation (task or data-dependency) is executed only after all the operations that precede it on the same architecture component are finished.

This worst-case computation is based on the knowledge of *maximal* duration of execution for each task, resp. data dependency, when it is scheduled onto some processor, resp. channel. The mapping d is constructed from this information in the straightforward manner. As an example, our scheduling in Figure 2 may give the minimal execution represented graphically in the figure at the left, where the duration of execution of each node in G is equal to the height of its corresponding box.



Formally, given a function $d : V \longrightarrow \mathbb{N}$, denoting the *duration of executing each scheduled task and data dependency*, we define two applications $\alpha, \beta : V \longrightarrow \mathbb{N}$, called resp. the *starting* and the *ending time*, by the two following equations:

$$\forall v \in V, \quad \alpha(v) = \max\big\{\beta(v') \mid (v', v) \in E\big\}, \quad \beta(v) = \alpha(v) + d(v) \qquad (1)$$

The computation of α and β lies on the notion of *topological ordering*, which says that each dag $G = (V, E)$ can be linearly ordered, such that each node has a bigger index than all of its predecessors in G. Placements that satisfy all the four requirements S1 to S4 of a plain scheduling, failing only on the requirement to be *acyclic*, cannot be executed: the system of equations (1) has no solution, which means that the schedule is deadlocked.

It is important to note that the way such a plain scheduling (hence non fault-tolerant) is obtained is outside the scope of our article. There exist numerous references on this problem, e.g. [21, 7, 9, 6, 13].

Before ending this section, let us remind the union operation on graphs: given two graphs $G = (V, E)$ and $G' = (V', E')$, their *union* is the graph $G'' = (V \cup V', E \cup E')$. Note that the two sets of nodes might not necessarily be disjoint, and in this case the nodes in the intersection $V \cap V'$ might be connected through edges that are in E or in E'.

3 Schedulings with replication

Our view of a static scheduling that tolerates *several* failure patterns is as a *family* of plain schedulings, one for each failure pattern which must be tolerated. This informal definition captures the simplest procedure one can design to obtain a fault tolerant scheduling: compute a plain scheduling for each failure pattern, then put all the plain schedulings together.

There may be several ways to put together the plain schedulings, each one corresponding to a decision mechanism for switching between them in the presence of failures. For example, we may have a table giving the sequence of operations to be executed on each processor in the presence of each failure pattern. But this choice corresponds to a centralised failure detection mechanisms. And, as we are in a distributed environment and therefore the information on the failure pattern is quite sparse, we would rather have some distributed mechanisms for propagating failure pattern information, based on a local decision mechanism.

To this end, we will assume that, in our family of plain schedulings, any two plain schedulings are not "contradictory" on the order they impose on each component in the architecture. That is, if in one plain scheduling G_1 a task a precedes a task b on the processor p, and both tasks are scheduled onto p in another plain scheduling G_2, then in G_2 these two tasks must be in the same order as on p. This assumption implies that, in the combination of all plain schedulings, each task or data dependency occurs at most once on each component in the architecture. This implies that we do not mask failures by rescheduling some operations and/or re-sending some data dependency. The propagation mechanism is then the impossibility to receive some data dependency within a bounded amount of time – that is, a timeout mechanism on each processor.

Consider, for example, the case of the task dag and the architecture graph given in Figure 1, and suppose that we want to tolerate either one failure for each channel, or one failure for each processor, with the exception of P_3 (say, an actuator whose replication cannot be supported at scheduling). Suppose also that task D must always be on processor P_3. We will consider the plain scheduling of Figure 2(c) (it supports the failure of P_2), and the plain schedulings of Figures 3(a) (for the failures of P_1 and/or C_1), and 3(b) (for the failures of P_4 and/or C_2). We will then combine these three plain schedulings into the *scheduling with replication* of Figure 3(c).

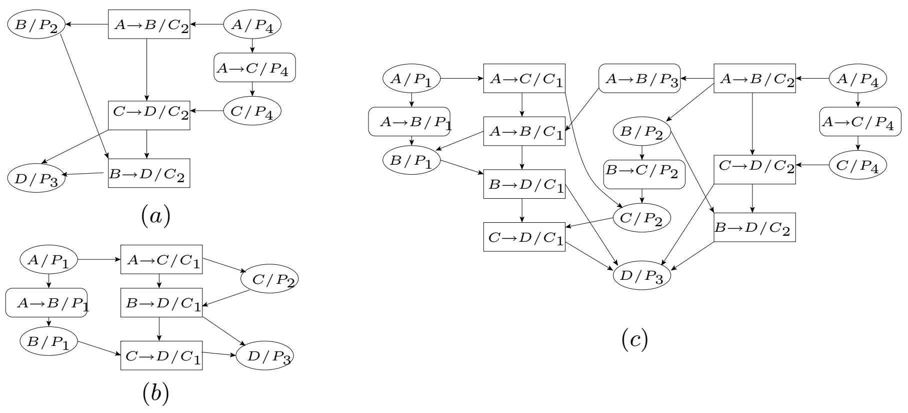


Fig. 3. (a) A plain scheduling tolerating the failures of P_1 and C_1. (b) A plain scheduling tolerating the failures of P_4 and/or C_2. (c) The scheduling with replication resulting from the combination of the two plain schedulings (a), (b) and the one from Figure 2.

This approach hides the following theoretical problem: the graphs that model such fault-tolerant schedulings *may contain circuits*. Formally, the process of "putting together" several plain schedulings must be modelled as a union of graphs, due to the requirement that, on each processor, each operation is scheduled at most once. But *unions of dags are not necessarily dags*. An example is provided in Figure 4. We have a symmetric "diamond" architecture, given in Figure 4(a), on which we want to tolerate two failure patterns: $\{C_1, C_5\}$ and $\{C_2, C_4\}$. We want to schedule the simple task dag of Figure 4(d), but with the constraint that A must be put on P_1 and B must be put on P_4 (say, because these processors are dedicated). The two failure patterns yield respectively the two reduced architectures of Figures 4(c) and (d). For both reduced architectures, we obtain respectively the plain schedulings of Figures 4(e) and (f). When combined, we obtain the scheduling with replication of Figure 4(g). More precisely, on the middle channel C_3, the data dependency $A\to B$ may flow in both directions! However none of the circuits obtained in the union of graphs really come into play in a real execution – they are there only because such a scheduling models the *set* of all possible executions.

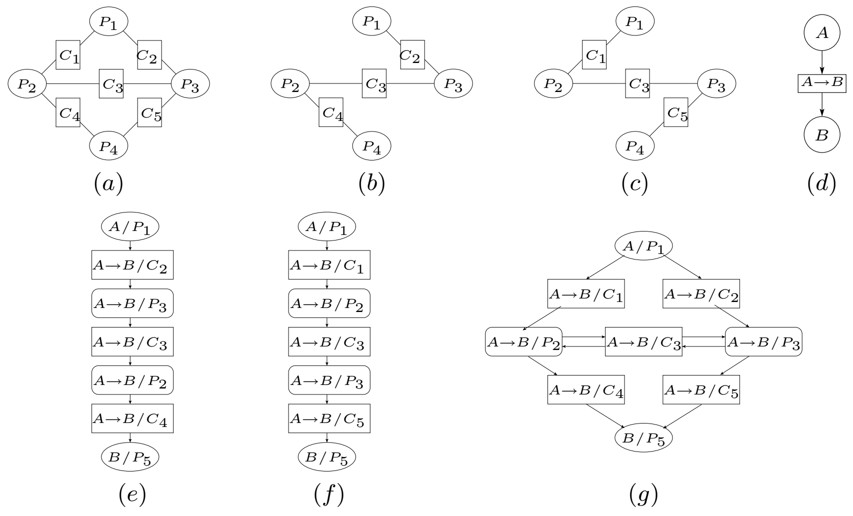


Fig. 4. (a) An architecture graph. $(b-c)$ The two reduced architectures corresponding to the failure patterns $\{C_1, C_5\}$ and $\{C_2, C_4\}$. (d) The task dag to be scheduled. $(e-f)$ The two plain schedulings of (d) onto (b) and (c) respectively. (g) The union of (e) and (f), which is not a dag!

The sequel of this section is dedicated to showing that the existence of circuits is not harmful when computing the execution of a scheduling with replication. We start with the formalisation of the notion of scheduling with replication.

Definition 3. *A **scheduling with replication** of a task dag $A = (O_A, D_A, E_A)$ is a labelled dag $G = (V, E, \lambda)$, with $\lambda : V \longrightarrow O_A \cup D_A$ satisfying the properties S1 and S2 of Definition 2, and the following additional property:*

S5 *For each circuit $v_1, \ldots, v_k, v_{k+1} = v_1$ of G, there exists $d \in D_A$ s.t. for all $1 \leq i \leq k$, $\lambda(v_i) = d$. That is, the only allowed circuits in G are those labelled with the same symbol, which must be in D_A.*

Since in schedulings with replication some operations might occur on several processors and some data dependencies might be routed through different paths to the same destination, the principles of execution stated in the previous section need to be relaxed. It is especially the first and the second principles that need to be restated as follows:

An **execution** of a plain scheduling onto a distributed architecture is governed by the following three principles:

P1' Each task can be executed only after it has received *at least one copy* of each data dependency it needs.
P2' Each data dependency can be transmitted only after *one of the replicas* of the tasks which may issue it has finished its execution, and hence has produced the corresponding data.
P3' Same as principle P3.

We use a mapping $d : V \longrightarrow \mathbb{N}$, which denotes the duration of executing each node of the task dag (resp. from O_A or D_A) onto each component of the architecture (resp. processor or communication link). We accept that some of the nodes may have zero duration of execution: it is the case with intra-processor data transmission; sometimes, even routings of some data dependencies between two adjacent channels may be considered as taking zero time. However we will consider that each complete routing of data through one or more channels takes at most one time unit. We will also impose a technical restriction on mappings d: for each circuit in G, if we sum up all the durations of nodes in the circuit, we get a strictly positive integer. This technical requirement will be essential in the proof of the existence of the minimal timeout mapping in the next section. Mappings $d : V \longrightarrow \mathbb{N}$ satisfying this requirement will be called *duration constraints*.

Definition 4. *An* ***execution*** *of a scheduling $G = (V, E, \lambda)$, with duration constraints $d : V \longrightarrow \mathbb{N}$, is a pair of functions $f = (\alpha, \beta)$, with $\alpha, \beta : V \longrightarrow \mathbb{N}$, respectively called starting time and ending time, and defined as follows:*

1. *Any task must start after it has received the first copy of each needed data dependencies: for each $a \in O_A$ and $v \in V$ with $(a, \lambda(v)) \in E_A$,*
$$\alpha(v) \geq \min \left\{ \beta(v') \mid \lambda(v') = a, (v', v) \in E \right\}$$
2. *Any data dependency starts after the termination of at least one of the tasks issuing it: for each $a \in D_A$ and $v \in V$ with $(a, \lambda(v)) \in E_A$,*
$$\alpha(v) \geq \min \left\{ \beta(v') \mid (v', v) \in E, \lambda(v') = \lambda(v) \text{ or } \lambda(v') = a \right\}$$
3. *Any task or data-dependency starts after the termination of all operations preceding it on the same architecture component:*
$$\text{for each } v, v' \in V \text{ with } (v', v) \in E \text{ and } (\lambda(v'), \lambda(v)) \notin E_A, \alpha(v) \geq \beta(v')$$
4. *The ending time of any given task or data-dependency is equal to its starting time plus its duration: for each $v \in V$, $\beta(v) = \alpha(v) + d(v)$.*

Executions can be compared componentwise, i.e., given two executions $f_1 = (\alpha_1, \beta_1)$ and $f_2 = (\alpha_2, \beta_2)$, we put $f_1 \leq f_2$ if, for each node $v \in V$, $\alpha_1(v) \leq \alpha_2(v)$ and $\beta_1(v) \leq \beta_2(v)$.

We may then prove that the set of executions associated to a scheduling with replication forms a *complete lattice* w.r.t. this order, that is, an *infimum* and a *supremum* exists for any set of executions. Definition 4 is in fact a fixpoint definition in the lattice of executions. But we have not yet showed that this lattice is nonempty – fact which would imply that there exists a least element in it, which will be called the **minimal execution**.

For plain schedulings, the existence of an execution is a corollary of the acyclicity of the graph G. As schedulings with replication may have cycles, we need to identify a weaker property that assures the existence of executions. The searched-for property is the following:

Definition 5. *A* ***pseudo-topological ordering*** *of a scheduling $G = (V, E, \lambda)$ is a bijection $\phi : V \longrightarrow [1 \ldots n]$ (where $n = card(V)$), such that for all $v \in V$, the following properties hold:*

1. *If $\lambda(v) \in O_A$, then for each $d \in D_A$ for which $(d, \lambda(v)) \in E_A$, there exists $v' \in V$ such that $\lambda(v') = d$, $(v', v) \in E$, and $\phi(v') < \phi(v)$.*
2. *If $\lambda(v) \in D_A$, then there exists $v' \in V$ with $(v', v) \in E$ and $\phi(v') < \phi(v)$, such that either $\lambda(v') = \lambda(v)$ or $(\lambda(v'), \lambda(v)) \in E_A$.*
3. *For each $v' \in V$, if $(v', v) \in E$, $(\lambda(v'), \lambda(v)) \notin E_A$, and $\lambda(v') \neq \lambda(v)$, then $\phi(v') < \phi(v)$.*

Lemma 1. *Any scheduling $G = (V, E, \lambda)$ has a pseudo-topological ordering .*

The proof of this lemma is constructive: we consider the following algorithm, whose entry is a scheduling $G = (V, E, \lambda)$ of a graph $A = (O_A, D_A, E_A)$ and whose output is a bijection $\phi_n : V \longrightarrow [1 \dots n]$ which represents a pseudo-topological ordering of G:

Algorithm 1 Pseudo-topological ordering
```
1 begin
2     $k := 1$; $X_0 := \emptyset$;
3     while $k \leq n$ do
4         Choose some node $w \in V$ such that:
              ⌘ For all $w'' \in V$ such that $(w'', w) \in E$, $\lambda(w'') \neq \lambda(w)$,
              and $(\lambda(w''), \lambda(w)) \notin E_A$, $\lambda(w'') \neq \lambda(w)$, we have $w'' \in X_{k-1}$;
              ⌘ If $\lambda(w) \in D_A$, then $\exists\ w' \in X_{k-1} \cap V$ with either $\lambda(w') = \lambda(w)$
              or $(\lambda(w'), \lambda(w)) \in E_A$ and such that $(w', w) \in E$;
              ⌘ If $\lambda(w) \in O_A$, then $\forall\ b \in D_A$ with $(b, \lambda(w)) \in E_A$, $\exists\ w' \in X_{k-1}$
              such that $\lambda(w') = b$ and $(w', w) \in E$;
5         $X_k := X_{k-1} \cup \{w\}$;
6         $\phi(k) := w$;
7         $k := k + 1$;
8     end while
9 end
```

Claim. The choice step at line 4 can be applied for any $k \leq card(V)$.

This claim can be proved by contradiction, since the impossibility to make the choice step in the algorithm would imply the existence of a circuit in the graph, whose nodes would be labeled with different symbols from $O_A \cup D_A$, in contradiction with requirement S5.

It follows that the algorithm terminates, that is, $X_n = V$. Therefore, the application $\phi : V \longrightarrow [1 \dots n]$ defined by $\phi(v) = k$ iff v is the node chosen at the k-th step, is a pseudo-topological order on V. This fact proves Lemma 1.

Theorem 1. *The set of executions associated to any scheduling $G = (V, E, \lambda)$ is nonempty.*

Proof. First, we use our Algorithm 1 to construct a pseudo-topological ordering $\phi : V \longrightarrow [1 \dots n]$ of the scheduling G. Then, we construct an execution $(\alpha^\phi, \beta^\phi)$ inductively as follows:

We start with the pair of totally undefined partial functions $\alpha_0^\phi, \beta_0^\phi : V \multimap [1 \dots n]$. That is, both $\alpha_0^\phi(v)$ and $\beta_0^\phi(v)$ are undefined for any node $v \in V$.

Suppose then, by induction, that we have built the pair of partial functions $(\alpha_{k-1}^\phi, \beta_{k-1}^\phi)$, such that $\alpha_{k-1}^\phi : V \multimap [1 \dots n]$ (resp. β_{k-1}^ϕ) associates to each vertex $v \in V$ with $\phi(v) \leq k-1$ the *starting* execution time (resp. the *ending* execution time). We then extend these partial functions to a new pair $(\alpha_k^\phi, \beta_k^\phi)$ with $\alpha_k^\phi, \beta_k^\phi : V \multimap\!\!\rightarrow [1 \dots n]$, by defining $\alpha_k^\phi(v_k)$ and $\beta_k^\phi(v_k)$ (where v_k is the k-th node of the pseudo-topological order, i.e., $\phi(v_k) = k$) as follows:

1. First, we compute the two following numbers:

$$x_1(v_k) = \max\big\{\beta_{k-1}^\phi(v') \mid (v', v_k) \in E \text{ and } (\lambda(v'), \lambda(v_k)) \notin E_A, \phi(v') < \phi(v_k)\big\}$$

$$x_2(v_k) = \begin{cases} \max\Big\{ \min\big\{\beta_{k-1}^\phi(v') \mid (v', v_k) \in E, \lambda(v') = b\big\} \mid \\ \qquad b \in D_A, (b, \lambda(v_k)) \in E_A, \phi(v') < \phi(v_k)\Big\} & \text{if } \lambda(v_k) \in O_A, \\ \min\big\{\beta_{k-1}^\phi(v') \mid (v', v_k) \in E, \phi(v') < \phi(v_k) \\ \quad \text{and either } \lambda(v') = \lambda(v_k) \text{ or } (\lambda(v'), \lambda(v_k)) \in E_A\big\} & \text{if } \lambda(v_k) \in D_A \end{cases}$$

   Note that, by the assumption that ϕ is a pseudo-topological ordering, all the sets involved in the above computations are nonempty.
2. We then take $\alpha_k^\phi(v_k) = \max\big(x_1(v_k), x_2(v_k)\big)$ and $\beta_k^\phi(v_k) = \alpha_k(v_k) + d(v_k)$.

It is then routine to check that α_n and β_n are indeed executions. □

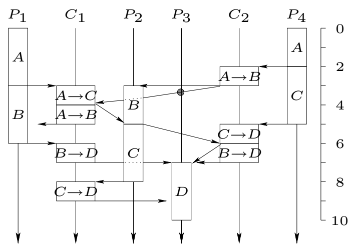
Corollary 1. *There exists a minimal execution associated to any scheduling $G = (V, E, \lambda)$.*

Proof. The minimal execution is defined as follows: for each $v \in V$,

$$\alpha_{min}(v) = \min\big\{\alpha_n^\phi(v) \mid \phi \text{ pseudo-topological ordering of } G\big\}$$
$$\beta_{min}(v) = \alpha_{min}(v) + d(v).$$

□

The figure here on the right gives the minimal execution for the scheduling in Figure 4 (in the absence of failures). The duration constraints are given by the vertical dimension of each rectangle, while the arrows without any target indicate lost arbitration between replicas.



4 Schedulings with replication and failures

The fault detection mechanism described in Section 3 uses timeouts on the scheduled task and data communications. This section is concerned with the formalisation of this mechanism. Again, we will make use of the notion of pseudo-topological ordering: it will help us in computing the timeouts. Note that computing the timeout of a scheduled operation as the max of the ending times of all the operations that precede it in the scheduling might not work, because of the possible existence of circular dependencies: this is the same problem emphasised in Section 2, and which required the introduction of the notion of pseudo-topological ordering.

A **failure pattern** in a scheduling $G = (V, E, \lambda)$ is just a subset $F \subseteq V$ of nodes. In general, we will consider a set of failure patterns $\mathcal{F} \subseteq \mathcal{P}(V)$. The result of removing a set F of nodes from a scheduling $G = (V, E, \lambda)$ is simply a labelled graph $G_F = (V_F, E_F, \lambda_F)$ with $E_F = E \cap (V_F \times V_F)$ and $\lambda_F = \lambda|_{V_F}$. G_F might not be a scheduling – some nodes may fail to satisfy the requirements S1 or S2 of Definition 3. This models the situation in which some operation does not receive one or more data that it needs. We will then say that G_F is a **scheduling reduced by the failure pattern** F.

If this graph *contains* a scheduling, i.e., there exists a scheduling $\overline{G}_F = (\overline{V}_F, \overline{E}_F, \overline{\lambda}_F)$ with $\overline{V}_F \subseteq V_F$ and $\overline{\lambda}_F = \lambda_F|_{\overline{V}_F}$, then we say that G **tolerates the failure pattern** F. Note that if G tolerates a failure pattern F, then it tolerates any failure pattern $F' \subseteq F$. On the other hand, if G tolerates two failure patterns F_1 and F_2, then in general it *does not* tolerate $F_1 \cup F_2$. Hence, what we give as failure patterns are *maximal* combination of faulty components.

Graphs like G_F cannot be "executed" using the principles given in Section 3 since some nodes may not receive all their data – we call such nodes as *starving*. Starving operations and data dependencies are not executed and, in order to prevent the system from being blocked, the component should try executing the next operation in its own scheduling. This is precisely how our obtained schedules mask the failures. Hence, components detect only a limited part of the "failure scenario", namely the subgraph which leads to the starving node.

We will formally say that even starving nodes are executed, but imposing that their duration of execution be zero. We will also require that, when an operation receives all its data (i.e., is not starving), it must be started before its timeout expires – just to avoid going into "philosophical" problems about the feasibility of several "instantaneous actions" in zero time:

Definition 6. *An* ***execution*** *of a scheduling reduced by a failure pattern $G_F = (V_F, E_F, \lambda_F)$, with duration constraints $d : V_F \longrightarrow \mathbb{N}$ and* timeout mapping *$\theta : V_F \longrightarrow \mathbb{N}$ is a tuple of functions $f = (\alpha, \beta, \varepsilon)$, with $\alpha, \beta : V_F \longrightarrow \mathbb{N}$ (the* starting, *resp.* ending times*) and $\varepsilon : V_F \longrightarrow \{0, 1\}$ (the* execution bit*), having the following properties:*

1. *For each $v \in V_F$, if we denote*
   $U_v^F = \{\beta(v') \mid v' \in V_F, (v', v) \in E_F, \text{ and } (\lambda(v'), \lambda(v)) \notin E_A, \lambda(v') \neq \lambda(v)\}$
   then $\alpha(v) \geq \max U_v^F$.
2. *For each $v \in V_F$ such that $\lambda(v) \in O_A$ and for each $a \in D_A$ such that $(a, \lambda(v)) \in E_A$, if we denote:*
   $$T_{v,a}^F = \{\beta(v') \mid v' \in V_F, \lambda(v') = a, (v', v) \in E_F, \text{ and } \varepsilon(v') = 1\}$$
   then $\alpha(v) \geq \min\big(\theta(v), \max_{a \in D_A}(\min T_{v,a}^F)\big)$.
3. *For each $v \in V_F$ such that $\lambda(v) \in D_A$, if we denote*
   $T_v^F = \{\beta(v') \mid v' \in V_F, \varepsilon(v') = 1 \text{ and either } (\lambda(v'), \lambda(v)) \in E_A \text{ or } \lambda(v') = \lambda(v)\}$
   then $\alpha(v) \geq \min\big(\theta(v), \min T_v^F\big)$.
4. *For each $v \in V_F$, $\beta(v) = \alpha(v) + d(v) \cdot \varepsilon(v)$ where* $\varepsilon(v) = \begin{cases} 1 & \text{if } \alpha(v) < \theta(v) \\ 0 & \text{otherwise} \end{cases}$

Using the notion of pseudo-topological ordering (with slight modifications), we may prove that, for each timeout mapping θ, the set of executions of G_F is nonempty. Moreover, we may prove that the **minimal execution** of G_F, denoted $(\alpha_F, \beta_F, \varepsilon_F)$ exists and can be computed as the least fix point of the following system of equations (here, minimality refers only to the tuple (α_F, β_F)):

1. For each $v \in V_F$ such that $\lambda(v) \in O_A$ and for each $a \in D_A$ such that $(a, \lambda(v)) \in E_A$, $\alpha_F(v) = \max\Big(\min\big(\theta(v), \max_{a\in D_A}(\min T^F_{v,a})\big), \max U^F_v\Big)$
2. For each $v \in V'$ with $\lambda(v) \in D_A$, $\alpha_F(v) = \max\Big(\min\big(\theta(v), \min T^F_v\big), \max U^F_v\Big)$.
3. And for each $v \in V$, $\beta_F(v) = \alpha_F(v) + d(v) \cdot \varepsilon_F(v)$, where $\varepsilon_F(v) = 1$ if $\alpha_F(v) < \theta(v)$, $\varepsilon_F(v) = 0$ otherwise.

Here $T^F_{v,a}$, T^F_v, and U^F_v are the notations from Definition 6.

However, a bad choice of a timeout mapping may induce that some operations that may receive all their data be not executed, because the wait for the reception of the respective data is greater than the timeout. We will therefore call an execution *correct* if this situation does not occur, that is:

Definition 7. *An execution $(\alpha, \beta, \varepsilon)$ of G_F is **correct** provided that it satisfies the following property: for each $v \in V_F$, if $\max\limits_{a\in D_A}(\min T^F_{v,a}) \neq \infty$ then $\varepsilon(v) = 1$.*

Then, the **minimal timeout** ensures that the minimal execution of the scheduling $G = (V, E, \lambda)$ reduced by any of the failure patterns from $\mathcal{F}$ is correct. We will construct this minimal timeout by repeatedly computing the minimal execution of each G_F in which non-starving nodes have an infinite timeout, whereas starving nodes share the same timeout for different failure patterns F.

Our algorithm starts with a zero timeout for all nodes, $\theta_0(v) = 0$. At each step i and for each failure pattern F, we utilize the timeout mapping $\overline{\theta}^F_i$ defined by $\overline{\theta}^F_i(v) = \theta_{i-1}(v)$ if v is starving, and $\overline{\theta}^F_i(v) = \infty$ otherwise. We compute the minimal execution $(\alpha_F, \beta_F, \varepsilon_F)$ induced by this timeout mapping, and then we compute $\theta_i(v)$ as the maximum of the starting times for v in each reduced graph G_F in which v is non-starving. In the following we denote $\overline{G}_F = (\overline{V}_F, \overline{E}_F, \overline{\lambda}_F)$ the *maximal* scheduling included in the graph G_F, assumed to exist for all $F \in \mathcal{F}$.

Algorithm 2 Timeout mapping computation

```
1  begin
2      forall $v \in V$ do $\theta_0(v) := 0$;
3      $i := 0$;
4      repeat
5          $i := i + 1$;
6          forall $F \in \mathcal{F}$ do
7              put $\overline{\theta}^F_i(v) = \begin{cases} \theta_{i-1}(v) & \text{if } v \notin \overline{V}_F \text{ (i.e., } v \text{ is starving)} \\ \infty & \text{otherwise} \end{cases}$
8              $(\alpha^F_i, \beta^F_i, \varepsilon^F_i)$ := the minimal execution induced by $\overline{\theta}^F_i$;
9          end for
10         forall $v \in V$ do $\theta_i(v) := \max_{F\in\mathcal{F}}\{\alpha^F_i(v) + 1 \mid v \in V_F\}$.
11     until $\theta_i = \theta_{i-1}$
12 end
```

Proposition 1. *For each $i \in \mathbb{N}$, $\theta_i \geq \theta_{i-1}$.*

The proof runs by induction on i, using the fact that mins and maxs commute with inequalities and that for each $F \in \mathcal{F}$, $\overline{\theta}_0^F$ is the smallest timeout mapping for which $\theta(v) = \infty$ for non-starving nodes v.

Proposition 2. *Algorithm 2 terminates, i.e., there exists $i \in \mathbb{N}$ with $\theta_i = \theta_{i-1}$.*

Proof. Remind that a path in a graph is called *simple* if it has no loop. We call the *weight* of a path $p = (v_1, \ldots, v_k)$ the sum $w(p) = \sum_{1 \leq i \leq k} d(v_i)$. For each $v \in V$, denote $M(v)$ the max of the weights of all simple paths in G ending in v.

We may prove by induction on i that $\beta_i^F(v) \leq M(v)$ for each $v \in V$, fact which would end our proof since the mapping M would then provide an upper bound for the increments on θ_i. To this end, denote $p = (v_1, \ldots, v_k)$ the path that ends in some node $v = v_k$ and that is defined by the property that $\beta_i^F(v_j) = \alpha_i^F(v_{j-1})$. Hence, p consists of nodes on a "critical path" that ends in v.

Two cases may occur: either v_1 has no predecessor in V or is a starving node, hence is "executed" (actually, skipped) when its timeout expires. No other case is possible since $(\alpha_i^F, \beta_i^F, \varepsilon_i^F)$ is a minimal execution.

In the first case, the claim is trivially proved, since p is a path that does not contain any circuit (again by definition of an execution).

In the second case, we will assume that v_1 is the only starving node in p and, moreover, that there does not exist another non-starving node v' which leads us to the first case. We have that

$$\alpha_i^F(v_2) = \beta_i^F(v_1) = \alpha_i^F(v_1) = \theta_{i-1}(v_1) = \alpha_{i-1}^{F'}(v_1) + 1$$

for some failure pattern F'. Since we have considered that the duration of any node is greater than 1 time unit, we then have that $\alpha_i^F(v_2) = \beta_i^F(v_1) \leq \beta_{i-1}^{F'}(v_1)$. On the other hand, by the induction hypothesis, $\beta_{i-1}^{F'}(v_1) \leq M(v_1)$. Therefore,

$$\beta_i^F(v) = \beta_i^F(v) - \alpha_i^F(v_2) + \beta_i^F(v_1) \leq w(p) + M(v_1)$$

Observe now that we must have $(\lambda(v_1), \lambda(v_2)) \notin E_A$. This follows from the assumption that v_1 is starving, v_2 is non-starving, and there exists no other non-starving node v' with $\beta_i^F(v') = \alpha_i^F(v_2)$. But then, the concatenation of any simple path that ends in v_1 with p cannot have circuits by requirement S5 in the definition of schedulings with replication. This means that $w(p)+M(v_1) \leq M(v)$, fact which ends our proof. □

Proposition 3. *The timeout mapping θ_i which is computed by the algorithm is correct for all failure patterns and is the minimal timeout mapping with this property.*

Proof. The correctness of the timeout mapping is ensured by the exit condition of the loop – we may only exit the loop when, for each node, the computed timeout $\theta_i(v)$ exceeds all the starting times in each failure pattern, and hence only starving nodes may reach their timeout. The second property follows by showing, by induction on i, that for each correct execution $(\alpha, \beta, \varepsilon)$ we have $(\alpha_i^F, \beta_i^F) \leq (\alpha, \beta)$. □

As an example, for the scheduling with replication given in Figure 3, the minimal timeout mapping is the following:

$\theta(A/P_1) = \theta(A/P_4) = 1$ $\quad$ $\theta(C/P_4) = \theta(A{\to}B/C_2) = 3$

$\theta(B/P_1) = \theta(B/P_2) = \theta(A{\to}C/C_2) = 4$ $\quad$ $\theta(A{\to}B/C_1) = 5$

$\theta(C/P_2) = \theta(C{\to}D/C_2) = 6$ $\quad$ $\theta(B{\to}D/C_1) = \theta(B{\to}D/C_2) = 7$

$\theta(C{\to}D/C_1) = \theta(D/P_3) = 9$

5 Conclusions

We have presented some results on the fundamentals of a theory of static fault-tolerant scheduling. The main problem we have investigated is the presence of circuits in unions of plain schedulings. We have introduced a notion of scheduling with replication, which models the loading of a system which is supposed to tolerate a family of failure patterns. We have also introduced the notion of execution of such a scheduling, which models some natural principles concerning the arbitration between replicas. We have also provided the notions of pseudo-topological ordering and of timeout mapping, which assure the existence and the correctness of minimal executions of a scheduling with replication.

Our work is only on the theoretical side, and the complexity of the algorithms in our paper is rather important for a practical use. In fact, all the problems treated here are NP-complete problems [8]. But we think that this theoretical layout may help developing heuristics for special architectures and/or special types of task dags, as it has been done in [18], and eventually compare the static scheduling performances with other techniques of fault-tolerant scheduling which take into account both processor and channel failures. Another possible approach for the fault-tolerant scheduling problem, that can tolerate both processor and channel failures, can be the combination of the redundancy techniques for processor failures with the fault-tolerant routing techniques. However, this combination seems not to have been given attention up to now.

References

1. M. Baleani, A. Ferrari, L. Mangeruca, M. Peri, S. Pezzini, and A. Sangiovanni-Vincentelli. Fault-tolerant platforms for automotive safety-critical applications. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES'03*, San Jose, USA, November 2003. ACM.
2. J. Bannister and K. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.
3. B. Bollobas. *Modern Graph Theory.* Graduate Texts in Mathematics. Springer Verlag, 1998.
4. A. Chowdhury, O. Frieder, E. Burger, D.A. Grossman, and K. Makki. Dynamic Routing System (DRS): Fault tolerance in network routing. *Computer Networks*, 31:89–99, 1999.
5. C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *Proceedings of 9th Euromicro Workshop PDP'2001*, pages 410–417. IEEE Computer Society Press, 2001.

6. G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *Euromicro Workshop on Real-Time Systems, EWRTS'97*, Toledo, Spain, June 1997. IEEE Computer Society Press.
7. M.R. Garey and D.S. Johnson. Complexity bounds for multiprocessor scheduling with resource constraints. *SIAM J. Computing*, 4(3):187–200, 1975.
8. M.R. Garey and D.S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman Company, San Francisco, 1979.
9. A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.
10. S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems*. Phd thesis, University of Pittsburgh, 1996.
11. S. Ghosh, R. Melhem, D. Mossé, and J. Sansarma. Fault-tolerant, rate-monotonic scheduling. *Real-Time Systems Journal*, 15(2), 1998.
12. A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *International Conference on Dependable Systems and Networks, DSN'03*, San-Francisco, USA, June 2003. IEEE.
13. T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.
14. H. Kopetz. TTP/A - the fireworks protocol. Research Report 23, Institut für Technische Informatik, Technische Universität Wien, Wien, Austria, 1994.
15. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS Approach. *MICRO*, 9:25–40, 1989.
16. Y. Oh and S.H. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems*, 7:315–330, 1993.
17. Y. Oh and S.H. Son. Scheduling hard real-time tasks with tolerance of multiple processor failures. Technical Report CS–93–28, Unversity of Virginia, May 1993.
18. C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Design, Automation and Test in Europe, DATE'04*, Paris, February 2004. IEEE.
19. K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. on Parallel and Distributed Systems*, 6:412–420, 1995.
20. F.B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2:145–154, 1984.
21. J.D. Ullman. Polynomial complete scheduling problems. In *Fourth ACM Symposium on Operating System Principles*, pages 96–101, New-York, USA, 1973.
22. L. Zakrevsky and M.G. Karpovsky. Fault tolerant message routing for multiprocessors. In J. Rolim, editor, *Parallel and Distributed Processing*, pages 714–731. Springer Verlag, 1998.