A scheduling heuristics for distributed real-time embedded systems tolerant to processor and communication media failures

A. Girault, H. Kalla

INRIA Rhône-Alpes Research Unit

ZIRST - 655 avenue de l’Europe - Montbonnot

38334 Saint-Ismier cedex, FRANCE

{alain.girault,hamoudi.kalla}@inrialpes.fr

Y. Sorel

INRIA Rocquencourt Research Unit

B.P.105 - 78153 Le Chesnay Cedex - FRANCE

yves.sorel@inria.fr

Abstract

Hardware fault tolerance is an important consideration in critical distributed real-time embedded systems that has been extensively researched. In these systems, critical real-time constraints must be satisfied even in the presence of hardware component failures. Our goal is to propose a solution to automatically produce a fault tolerant distributed schedule of a given algorithm onto a given distributed architecture, according to real-time constraints. The distributed architectures we consider have bidirectional point-to-point communication links. Our solution is a list scheduling heuristics, based on disjoints paths to tolerate a fixed number of arbitrary processor and communication link failures. Because of the resource limitation in embedded systems, our heuristics implements a software solution based on the active replication technique, where each operation of the algorithm is replicated on different processors. Through a detailed example, we show the techniques used to satisfy the real-time constraints and to tolerate the failures of processors and communications links. Simulations show the efficiency of our method compared to other heuristics found in the literature.

Keywords

Safety critical systems, embedded real-time systems, distributed architectures, fault tolerant systems, static scheduling heuristics.

1 Introduction

Embedded real-time systems are being increasingly used in a major part of critical applications such as avionics, automotive, nuclear, robotics, and telecommunication. In these systems, critical real-time constraints must be satisfied [14, 42], since timing constraints which are not met may involve a system failure leading to a human, ecological, and/or financial disaster. One of the major problems of these systems is dependability [4, 45], since the malfunction or the failure of system's components (hardware or software) can lead to a catastrophe. The dependability of such real-time systems can be increased through *hardware* or *software fault tolerance* techniques [46], such that a system built with fault tolerance capabilities will keep operating even in the presence of failures [31]. Hardware fault tolerance improves the dependability of distributed real-time systems by redundancy: adding extra hardware (processors, communication media, actuators, sensors) [11, 34] or extra software (tasks, messages) [30, 23] into the system. However, in most embedded systems hardware fault tolerance techniques, based on hardware redundancy, are not preferred due to the limited resources available, for reasons of weight, encumbrance, energy consumption (e.g., autonomous vehicles), radiation resistance (e.g., nuclear or space), or price constraints (e.g., consumer electronics). Therefore, critical embedded systems increasingly use *software redundancy* to achieve the required dependability.

The general domain of our research is hardware fault tolerance, based on software redundancy, in distributed critical embedded systems. Our ultimate goal is to develop new scheduling heuristics to produce automatically a fault tolerant distributed code from a given specification of the desired system. Concretely, we are given as input a specification of the algorithm to be distributed ($\mathcal{A}lg$), a specification of the target distributed architecture ($\mathcal{A}rc$), some distribution constraints ($\mathcal{D}is$), some information about the execution times of the algorithm operations on the processors and the communication times of the algorithm data-dependencies on the communication links ($\mathcal{E}xe$), some real-time constraints ($\mathcal{R}tc$), and a fixed number of components failures to be tolerated ($\mathcal{N}cf$); components are processors *and* communication links. Our goal is to find a static schedule of $\mathcal{A}lg$ onto $\mathcal{A}rc$, satisfying $\mathcal{D}is$, and tolerant to at most $\mathcal{N}cf$ component failures, with an indication whether or not this schedule satisfies $\mathcal{R}tc$ w.r.t. $\mathcal{E}xe$. The global picture of our methodology is shown in Figure 1. In this paper, we focus on the distribution algorithm.

Finding an algorithm that gives the *best* fault tolerant schedule w.r.t. the execution times is a well-known NP-hard problem [16]. Instead, we propose a heuristics that gives *one* scheduling, possibly not the best. This heuristics takes into account the execution time of *both* the computation operations and the data communications to optimize the critical path of the obtained schedule. Operations scheduled on the distributed heterogeneous architecture are guaranteed to complete even in the presence of a specified number of components failures at any instant of time. All the components are assumed to be fail-silent. But there is no need for a complex failure detection mechanism; and there is no need for the healthy processors to propagate the state of the faulty ones. The strategy used to schedule operation replicas ensures a minimum run-time overhead in the faulty system (a system presenting at least one component failure) compared to the nominal one (with no failures).

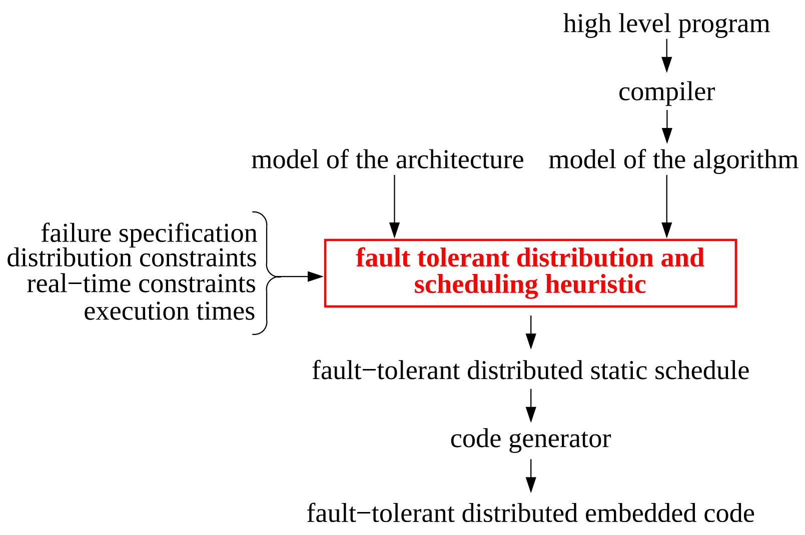
high level program
compiler
model of the architecture
model of the algorithm
failure specification
distribution constraints
real−time constraints
execution times
fault tolerant distribution and scheduling heuristic
fault−tolerant distributed static schedule
code generator
fault−tolerant distributed embedded code

Figure 1: Global picture of our methodology

The paper is organized as follows. Section 2 describes the related work. Section 3 states our fault tolerance problem and presents the various models used by our method. Section 4 presents the proposed heuristics for providing fault tolerance for architecture with multiple processors linked by a set of point-to-point links. Section 5 explains the runtime behavior of the fault-tolerant schedules produced by our heuristics. Section 6 presents the analysis of the performance of our heuristics and the simulations results. Finally, section 7 concludes and proposes directions for future research.

2 Related work

The key to building fault tolerant distributed real-time systems in general is redundancy: redundant components and/or algorithmic blocks are added to the system. There are three classes of redundancy: hardware, software, and time redundancy [30]. As we are targeting embedded systems with limited hardware resources, in this section we only present work involving software and time redundancy.

In the literature, we can identify several software fault tolerance approaches for dependable embedded systems, tolerating: only processors failures, only communication media failures, or both processors and communication media failures.

The fi rst category of approaches tolerates only processor failures. Several heuristics for scheduling real-time tasks in multiprocessors architectures have been proposed. They are based either on *active software replication* [9, 10, 17, 29] or *passive software replication* [36, 2, 39, 7, 3, 35]. In the active replication technique, multiple redundant copies of each task are scheduled on different processors, which are run in parallel to tolerate a fi xed number of processor failures. For instance, Hashimoto et al. propose a scheduling algorithm to tolerate a single processor failure [29]: each task is actively replicated on two different processors. However, the proposed algorithm is limited to homogeneous systems (the execution characteristics of a given task are identical on *all* processors) and fully connected architectures. Our work is more general since we target heterogeneous architectures, and we only require a minimum number of disjoint paths between any two processors.

In the passive replication strategy, also called primary backup approach, each task is replicated into several copies: one primary and several backup copies; only the primary replica executes; when the primary fails, one of the backups is selected to become the new primary. For instance, Xiao proposes a fault tolerant real-time scheduling algorithm that can tolerate one processor failure in a heterogeneous distributed system [40]. Faults are assumed to be permanent and the approach describes a primary backup scheme, where each real-time task has two copies. In [36], Oh and Son propose a one timely fault tolerant scheduling algorithm to tolerate one processor failure and to minimize the obtained schedule length. They assume that processors are fail-stop and that the failure of a processor can be detected by the other processors. The backup copies scheduled on the same processor are overlapped with each other in time, in order to reduce the fault tolerance overhead.

The second category of approaches tolerates communication media failures. Several techniques have been proposed, either *proactive* or *reactive*. In the proactive scheme [15, 33, 41, 32, 13], multiple redundant copies of a message are sent along disjoints paths. In the reactive scheme [44, 28, 27, 8, 5], one copy of the message, called primary, is sent, and if the primary copy fails, another copy of the message, called backup, will be transmitted.

The last category of approaches tolerates both processors and communication media failures. In [38, 48, 24], failures are tolerated using the fault recovery scheme and a primary/backups strategy. In [12], Dima et al. propose an original off-line fault tolerant scheduling algorithm which uses the active replication of tasks and communications to tolerate a set of failure patterns; each failure pattern is a set of processor and/or communications media that can fail simultaneously, and each failure pattern corresponds a reduced architecture. The proposed algorithm starts by building a basic schedule for each reduced architecture plus the nominal architecture, and then merges these basic schedules to obtain a distributed fault tolerant schedule. It has been implemented very recently by Pinello et al. [37]. In the future, we plan to compare this method with ours.

Like the other researchers belonging to the last category, we propose an automatic solution to the fault tolerance distributed problem. The conjunction of the four following points makes our approach original:

1. We take into account the execution time of both the computation operations and the data communications to optimize the critical path of the obtained schedule.

2. Since we produce a static schedule, we are able to compute the expected completion date for any given operation or data communication, both in the presence and in the absence of failures. Therefore we are able to check the real-time constraints $\mathcal{R}tc$ before the execution. If $\mathcal{R}tc$ is not satisfi ed, we can give a warning to the designer, so that he can decide whether to add more hardware or to relax $\mathcal{R}tc$.

3. The source algorithm $\mathcal{A}lg$ can be designed with a high-level programming language based on a formal mathe-

matical semantics. For instance, this is the case of synchronous languages, which are moreover well suited to the programming of embedded critical systems [26, 6]. The advantage is that $\mathcal{A}lg$ can be formally verified with model-checking and theorem proving tools, and therefore we can assume safely that it is free of design faults. The scheduling method we propose in this paper preserves this property.

4. Operations scheduled on the distributed architecture are guaranteed to complete if at most $\mathcal{N}cf$ processors or communication links fail at any instant of time. There is no need for a complex failure detection mechanism. Finally, due to the scheduling strategy used, the time needed for handling a failure is minimal.

The heuristics presented in this paper is our most recent work for integrating fault tolerance in the SYNDEX [1] tool [22], a system level CAD software tool for optimizing the implementation of real-time embedded applications on multicomponent architecture. Prior work has been published in [20, 21, 12, 19, 18]. The method presented in this paper is more general since it tolerates *both* processors *and* communication links failures.

3 System models and assumptions

3.1 Architecture model

The architecture is modeled by a graph $\mathcal{A}rc$, where each vertex is a processor, and each edge is a bidirectional communication link. Classically, a processor is made of one computation unit, one local memory, and one or more communication units, each connected to one communication link. Communication units execute data transfers, called `comms`. The chosen communication mechanism is the send/receive [25], where the send operation is non-blocking and the receive operation blocks in the absence of data. Figure 2(b) is an example of architecture graph, with four processors P1, P2, P3 and P4, and four point-to-point communications links L12, L14, L23, and L34.

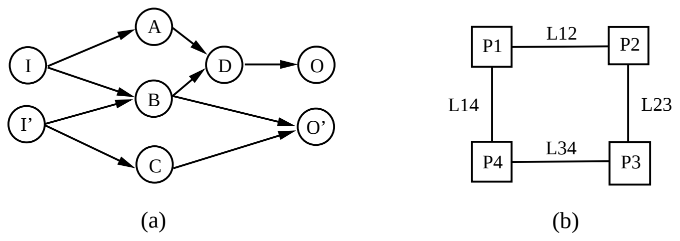

Figure 2: Example of (a) an algorithm graph $\mathcal{A}lg$ and (b) an architecture graph $\mathcal{A}rc$.

In the sequel, we note $\mathcal{P}$ the set of processors of $\mathcal{A}rc$. For Figure 2(b), $\mathcal{P}$=\{P1,P2,P3,P4\}.

[1]http://www-rocq.inria.fr/syndex

3.2 Algorithm model

The algorithm to be distributed is modeled by a *data-flow graph* $\mathcal{A}lg$. Each vertex is an *operation* and each edge is a *data-dependency*. The algorithm is executed repeatedly for each input event from the sensors (operations without predecessors) in order to compute the output events for the actuators (operations without successors). This periodic sampled model is commonly used for embedded systems and automatic control systems. Operations of $\mathcal{A}lg$ can be either:

- a computation operation (`comp`): its inputs must precede its outputs; the outputs depend only on the input values; there is no internal state variable and no other side effect;
- a memory operation (`mem`): the data is held in sequential order between iterations; the output precedes the input, like a register in Boolean circuits;
- an external input/output operation (`extio`); operations with no predecessor in the data flow graph (resp. no successor) are the external input interfaces (resp. output), handling the events produced by the sensors (resp. actuators).

Figure 2(a) is an example of algorithm graph, with eight operations: (I,I') are sensor operations, and (O, O') are actuator operations, while (A, B, C, D) are computation operations. The data-dependencies between operations are depicted by arrows. For instance, the data-dependency A $\triangleright$ D corresponds to the sending of some arithmetic result computed by A and needed by D.

		operation							
	time	I	I'	A	B	C	D	O	O'
proc.	P1	2.5	∞	2.5	3.0	2.0	1.5	3.0	3.0
	P2	1.5	1.5	1.5	2.0	1.0	0.5	2.0	∞
	P3	2.5	∞	2.5	3.0	2.0	1.5	3.0	3.0
	P4	1.5	1.5	1.5	2.0	1.0	0.5	2.0	∞

		data-dependency								
	time	I $\triangleright$ A	I $\triangleright$ B	I' $\triangleright$ B	I' $\triangleright$ C	A $\triangleright$ D	B $\triangleright$ D	C $\triangleright$ O'	B $\triangleright$ O'	D $\triangleright$ O
link	L12	1.0	2.0	1.5	2.0	1.5	2.0	1.5	1.0	1.5
	L23	2.0	4.0	3.0	3.0	4.0	4.0	3.0	2.0	3.0
	L14	1.0	2.0	1.5	2.0	1.5	2.0	1.5	1.0	1.5
	L34	2.0	4.0	3.0	3.0	4.0	4.0	3.0	2.0	3.0

Table 1: Distributed constraints $\mathcal{D}is$ and execution/transmission times $\mathcal{E}xe$ for operations and data-dependencies.

To each operation o of $\mathcal{A}lg$, we associate in a table $\mathcal{E}xe$ its execution time on each processor: each pair $\langle o, p \rangle$ of $\mathcal{E}xe$ is the worst case execution time (WCET) of the operation o on the processor p, expressed in time units. Since the target architecture is heterogeneous, the WCET for a given operation can be distinct on each processor. Similarly, to each data-dependency of $\mathcal{A}lg$, we associate in a table $\mathcal{E}xe$ its transmission time on each communication link: each pair $\langle d, l \rangle$ of $\mathcal{E}xe$ is the worst case transmission time (WCTT) of the data dependency d on the communication link l, again expressed in time units. Since the target architecture is heterogeneous, the WCCT for a given data-dependency can be distinct on each communication link.

For instance, $\mathcal{E}xe$ for $\mathcal{A}lg$ and $\mathcal{A}rc$ of Figure 2 is given in Table 1. The table only gives the transmission times for *inter-processor* communications. For an *intra-processor* communication, the time is always zero time unit.

Finally, specifying the distribution constraints $\mathcal{D}is$ amounts to associating the value '∞' to some pairs $\langle o, p \rangle$ of $\mathcal{E}xe$, meaning that o cannot be executed on p (see Table 1). The reason for this might be because the operation o requires a specifi c co-processor to execute, for instance.

3.3 Failure model

As said in the introduction, our goal is to fi nd an $\mathcal{N}cf$ fault tolerant static schedule of $\mathcal{A}lg$ onto $\mathcal{A}rc$, satisfying $\mathcal{D}is$ and $\mathcal{R}tc$. A fault tolerant static schedule is defi ned as a schedule in which no real-time constraints $\mathcal{R}tc$ are missed, despite $\mathcal{N}cf$ arbitrary component (processor and communication link) failures. The failures considered are fail-silent component failures (permanent or intermittent): each component exhibits only omission or crash failures. Since software redundancy makes the failure behavior of such system more predictable, masking faults (i.e., never showing the effect of faults [31]) by redundancy is the basic principle of our method.

We assume that all values returned by the replicas of any given input operation are identical in the same iteration. The real-time constraints $\mathcal{R}tc$ can be, for instance, a deadline for the completion date of the whole schedule. If the user wants to be more precise, he/she can specify a deadline on the completion date of a particular operation of $\mathcal{A}lg$. The fact that the obtained schedule is static allows the computation of any such completion date w.r.t. $\mathcal{E}xe$.

4 The proposed fault tolerant scheduling algorithm

In this section, we discuss the basic principles used in the proposed approach for tolerating component failures in architectures with point-to-point links, followed by a description of our algorithm.

4.1 Algorithm principles

Our algorithm is a list scheduling heuristics based on an *active replication strategy* and *disjoints paths*, which allows at least $\mathcal{N}cf$+1 replicas of an operation to be scheduled on different processors, which are run in parallel to tolerate at most $\mathcal{N}cf$ component failures.

We use the software redundancy of both `comps`/`mems`/`extios` and `comms`. Each operation of $\mathcal{A}lg$ graph is replicated on $\mathcal{R}$ different processors of the architecture graph, where $\mathcal{R} \geq \mathcal{N}cf + 1$. Each of the best $\mathcal{N}cf + 1$ replicas send their results in parallel to all the replicas of all the successor operations in the data-flow graph. Therefore, each operation will receive its set of inputs $\mathcal{N}cf + 1$ times via disjoints paths; as soon as it receives the fi rst set, the operation is executed and ignores the later inputs. In some cases, the replica of an operation will only receive some of its inputs *once*, through an intra-processor communication; this will occur whenever one of its predecessor operations has one of its replicas scheduled on the *same* processor.

For the sake of simplicity, suppose we have an operation A with only one input produced by its predecessor I (Figure 3(a)); suppose also that we want to tolerate one component failure ($\mathcal{N}cf$=1); then, both operations A and I will be actively replicated on two distinct processors (Figure 3(b)).

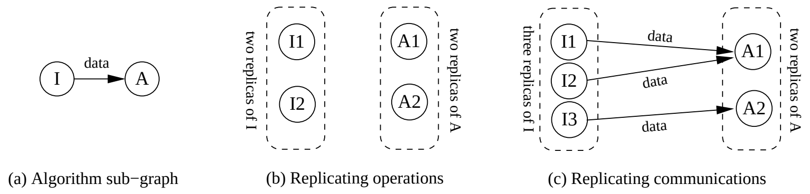

Figure 3: Software redundancy of operations and communications.

Consider the replicas I1 and I2 of I, which are assigned respectively to processors P1 and P3, as shown in the diagram of Figure 4. In this diagram, each operation is represented by a white box, whose height is proportional to its execution time. Each `comm` is represented by a gray box, whose height is proportional to its communication time, and whose ends are bound by two arrows: one from the source operation and one to the destination operation. In our example, each replica of A is assigned to a processor distinct from P1 and P3, so the `comm` from each replica of I to each replica of A will be actively implemented via disjoints paths as *inter-processor* communications; indeed, the communications I1 ▷ A1 and I2 ▷ A1 are implemented via two disjoint paths, and so are I1 ▷ A2 and I2 ▷ A2.

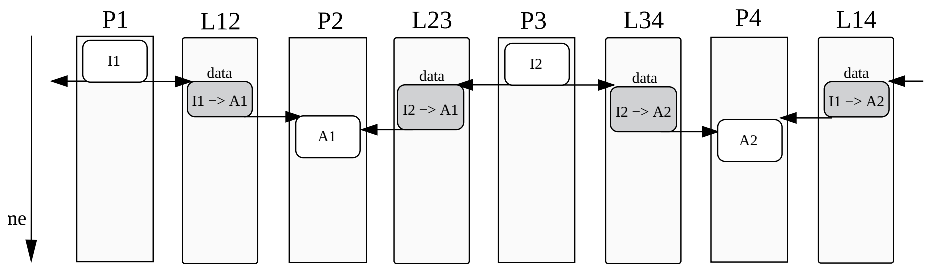

Figure 4: Schedule $\mathcal{N}cf$+1 replicas of each operation with $\mathcal{N}cf$=1.

Since the communication cost between operations assigned to the same processor is considered to be zero, replicating an operation more than $\mathcal{N}cf$+1 times can reduce the global interprocessor communication overheads of the schedule; this property is known as the locality of computations. For example, consider the schedule of Figure 4: if I is additionally replicated on P4, the temporary schedule length of processor P4 can be reduced, both in the presence and in the absence of component failures, as shown in Figure 5. Indeed, the `comm` from I to each replica of A will be implemented only once, as a single *intra-processor* communication. This situation corresponds to Figure 3(c).

Finally, the temporary schedule diagrams of Figures 5 and 4 can mask the failures of one arbitrary processor or communication link. For example, if the link L12 fails, then operation A1, scheduled on P2, will use the data received from replica I2 scheduled on P3, as shown in Figure 6.

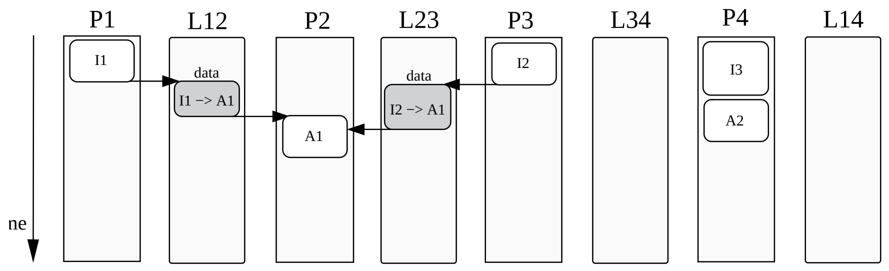

Figure 5: Schedule more than $\mathcal{N}cf$+1 replicas of an operation.

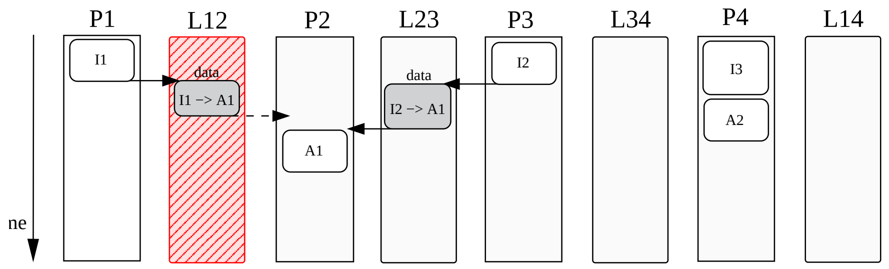

Figure 6: The temporary schedule diagram when link L12 fails.

4.2 Notations

Before describing our proposed scheduling heuristics (Figure7), we fi rst defi ne the following notations which are used in the rest of this paper. Our heuristics runs in a succession of steps. At each step, one operation is selected to be scheduled on a subset of processors. The superscript number in parentheses refers to the step of the heuristics:

- $O_{cand}^{(n)}$: The list of *candidate* operations, built from the algorithm graph vertexes. An operation is said to be a candidate if all its predecessors are already scheduled.
- $\mathcal{FT}_{sched}^{(n)}$: The list of already *scheduled* actions (operations, data-dependencies), with their respective component (processor or communication link) and start time.
- $pred(o_i)$: The set of predecessors of operation o_i.
- $succ(o_i)$: The set of successors of operation o_i.
- $R^{(n)}$: The critical path length.
- $E_{exc}^{(n)}(o_i, p_j)$: The end of execution time of operation o_i scheduled on processor p_j.

- $E^{(n)}_{com}(o_i, o_j)$: The end of data communication time from operation o_i to operation o_j.
- $\overline{S}^{(n)}(o_i)$: The latest start time from end of o_i, defined to be the length of the longest path from the outputs operations to o_i.
- $S^{(n)}_{best}(o_i, p_l)$: The earliest time at which operation o_i can start execution on processor p_l, computed as follows:

  $$S^{(n)}_{best}(o_i, p_l) = \max_{o_j \in pred(o_i)} \left\{ \min_{k=1}^{Npf+1} E^{(n)}_{com}(o^k_j, o_i) \right\}$$

  where o^k_j is the k^{th} replica of o_j. If o_i and o^k_j are scheduled in the same processor p_l, then

  $$E^{(n)}_{com}(o^k_j, o_i) = E^{(n)}_{exc}(o^k_j, p_l)$$

- $S^{(n)}_{worst}(o_i, p_l)$: The latest time at which operation o_i can start execution on processor p_l, taking into account all the predecessors replicas; it is computed as follows:

  $$S^{(n)}_{worst}(o_i, p_l) = \max_{o_j \in pred(o_i)} \left\{ \max_{k=1}^{Npf+1} E^{(n)}_{com}(o^k_j, o_i) \right\}$$

  where o^k_j is the k^{th} replica of o_j. If o_i and o^k_j are scheduled in the same processor p_l then

  $$E^{(n)}_{com}(o^k_j, o_i) = E^{(n)}_{exc}(o^k_j, p_l)$$

As a cost function for our greedy list scheduling heuristics, we use the *dependable schedule pressure*, noted $\delta^{(n)}$, in order to give priority between operations. It uses a *variant* $\tilde{\sigma}^{(n)}$ of the schedule pressure $\sigma^{(n)}$ defined in [43]. The schedule pressure $\sigma^{(n)}$ is computed, for each operation $o_i \in O^{(n)}_{cand}$ and each processor $p_j \subset \mathcal{P}$, as follows:

$$\sigma^{(n)}(o_i, p_j) := S^{(n)}_{best}(o_i, p_j) + \overline{S}^{(n)}(o_i) - R^{(n-1)} \tag{1}$$

Then, $\tilde{\sigma}^{(n)}$ is computed as follows:

$$\tilde{\sigma}^{(n)}(o_i, p_j) := S^{(n)}_{worst}(o_i, p_j) + \overline{S}^{(n)}(o_i) - R^{(n-1)} \tag{2}$$

where the use of $S^{(n)}_{worst}$ instead of $S^{(n)}_{best}$ allows the reduction of the schedule length overhead in the presence of $\mathcal{N}cf$ arbitrary component failures. Then, the dependable schedule pressure $\delta^{(n)}$ is computed for each operation $o_i \in O^{(n)}_{cand}$ and a set of processors $P_{opts} \subset \mathcal{P}$ (recall that $\mathcal{P}$ is the set of all processor) as follows:

$$\delta^{(n)}(o_i, P_{opts}) := \max_{p_j \in P_{opts}(o_i)} \tilde{\sigma}^{(n)}(o_i, p_j) \tag{3}$$

where

$$P_{opts}(o_i) := \left\{ p_j | p_j \in \min_{p_j \in \mathcal{P}}^{\mathcal{N}cf+1} \tilde{\sigma}^{(n)}(o_i, p_j) \right\} \tag{4}$$

The dependable schedule pressure measures how much the scheduling of an operation lengthens the critical path of the algorithm in the absence of component failures.

4.3 Scheduling algorithm

Our heuristics is a greedy list scheduling [47], called the improved Fault Tolerance Based Active Replication strategy (iFTBAR) algorithm [2]. Initially, $\mathcal{FT}^{(0)}_{sched}$ is empty and $O^{(0)}_{cand}$ is the list of operations without any predecessors. At the n-th step ($n \geq 1$), the list of *already scheduled* operations $\mathcal{FT}^{(n)}_{sched}$ is kept. Also, the list of *candidate* operations $O^{(n)}_{cand}$ is built from the algorithm graph vertexes.

```
The iFTBAR Algorithm:
Inputs: $\mathcal{A}lg$, $\mathcal{A}rc$, $\mathcal{E}xe$, $\mathcal{R}tc$, $\mathcal{D}is$, $\mathcal{N}cf$;
Output: a fault-tolerant distributed schedule $\mathcal{FT}_{sched}$;
begin
Initialize the lists of candidate and scheduled operations:
  n := 0;
  $O^{(0)}_{cand} := \{o \in O \mid pred(o) = \emptyset\}$;
  $\mathcal{FT}^{(0)}_{sched} := \emptyset$;
while $O^{(n)}_{cand} \neq \emptyset$ do
    ① Compute the modified schedule pressure $\tilde{\sigma}^{(n)}$ for each operation $o_{cand}$ of $O^{(n)}_{cand}$ on each processor $p_j$ of $\mathcal{P}$ using Equation (2).
    ② Compute the set of optimal processors $P_{opts}$ for each candidate operation $o_{cand}$ of $O^{(n)}_{cand}$ using Equations (3) and (4).
    ③ Select the best candidate operation $o_{best}$, such that:
       $\delta^{(n)}_{urgent}(o_{best}) := \max_{o_{cand} \in O^{(n)}_{cand}} \delta^{(n)}(o_{cand}, P_{opts})$;
    ④ Schedule $\mathcal{N}cf$+1 replicas of the best candidate operation $o_{best}$ on each processor of $P_{opts}$ computed at micro-step ②.
       The comms implied by this scheduling decision are also scheduled here, such that each replica of $o_{best}$ receives its input data via disjoints links from only the best $\mathcal{N}cf$+1 replicas of each of these predecessor operations.
    ⑤ Try to reduce the worst start time $S^{(n)}_{worst}$ of each replica of $o_{best}$ by applying the procedure "Minimize Start Time" (see Figure 8).
    ⑥ Update the lists of scheduled and candidate operations :
       $\mathcal{FT}^{(n+1)}_{sched} := \mathcal{FT}^{(n)}_{sched} \cup_{k=1}^{\mathcal{N}cf+1} \{\langle o^k_{best}, P_{opts}(k), S_{best}(o^k_{best}, P_{opts}(k))\rangle\}$;
       $O^{(n+1)}_{cand} := O^{(n)}_{cand} - \{o_{best}\} \cup \{o' \in succ(o_{best}) \mid pred(o') \subseteq \mathcal{FT}^{(n+1)}_{sched}\}$;
    ⑦ n := n + 1;
end while
return the fault-tolerant distributed schedule $\mathcal{FT}^{(n)}_{sched}$;
end
```

Figure 7: The fault tolerant scheduling algorithm iFTBAR.

[2] FTBAR was first presented in [18, 19]. The difference with iFTBAR is that it tolerates only *processor* failures.

At each step n, one operation of the list $O_{cand}^{(n)}$ is selected to be scheduled. To select an operation, we select at the micro-steps ① and ②, for each candidate operation o_{cand}, the $\mathcal{N}cf$+1 optimal processors P_{opts} having the minimum schedule pressure. Then, among those pairs $\langle o_{cand}, P_{opts}\rangle$, we select at the micro-step ③ the best one $\langle o_{best}, P_{opts}\rangle$ having the maximum schedule pressure, i.e., the most urgent pair.

The selected operation o_{best} is replicated and scheduled at the micro-step ④ on each processor of $P_{opts}(o_{best})$, and the `comms` implied by these scheduling decisions are also scheduled here via disjoint paths. At this micro-step, the start time of each replica of the selected operation o_{best} is possibly reduced by replicating its predecessors using the procedure *Minimize Start Time*, which is an extended procedure of the one proposed in [1] (see Figure 8).

```
The Minimize Start Time Procedure (MSTP):
Inputs: an operation $o$, a processor $p$, $S_{worst}$, a schedule $\mathcal{FT}_{sched}$ ;
Output: an improved schedule $\mathcal{FT}_{sched}$;
begin
    ❶ if ($o$ cannot be scheduled on $p$) or ($o$ is an input operation) then return $\mathcal{FT}_{sched}$;
    ❷ Find out the Latest Immediate Predecessor (LIP) of $o$;
    ❸ Minimize the start time of this LIP by recursively calling this procedure:
          $\mathcal{FT}_{sched}$ := MSTP$\langle$LIP$, p, S_{worst}(LIP, p), \mathcal{FT}_{sched}\rangle$;
    ❹ Compute the $new\ S_{worst}(o, p)$;
    ❺ if ( $new\ S_{worst}(o, p) \geq S_{worst}(o, p)$ ) then
          • Undo all the replications just performed in ❸;
          • Reschedule $o$ to $p$ at $S_{best}(o, p)$; the comms implied by this schedule are also scheduled here.
          • return $\mathcal{FT}_{sched}$;
    ❻ else
          • $\mathcal{FT}_{sched} := \mathcal{FT}_{sched} \cup \{\langle$LIP$, p, S_{best}($LIP$, p)\rangle\}$;
          • Find out the new LIP of $o$ and goto ❸;
end
```

Figure 8: A procedure to minimize the start time of an operation.

For each replica o_{best}^k of the selected candidate operation o_{best} and for each pair $\langle o_j, o_{best}^k\rangle$ where $o_j \in pred(o_{best})$, the data dependency $\langle o_j, o_{best}^k\rangle$ is implemented as $\mathcal{N}cf$+1 `comms` assigned to as many disjoint paths. However, if there exists one replica of o_j implemented on the same processor as o_{best}^k, then no `comms` are added since the data dependency is an intra-processor communication (see Section 4.1 and Figure 5).

When a `comm` is added, it is assigned to the set of communication units bound to the communication media connecting the processors executing the source and destination operations. At the end, all the `comms` assigned to the same communication unit are statically scheduled. The `comms` are thus totally ordered over each communication medium. Provided that the network preserves the integrity and the ordering of messages, this total order of the `comms` guarantees that data will be transmitted correctly between processors. The obtained schedule also guarantees a deadlock free execution [22]. Also, the strategy used to schedule operations ensures a minimum run-time overhead in the faulty system (a system presenting at least one arbitrary component failure) by using $S^{(n)}_{worst}(o,p)$ to give priority to operations and $S^{(n)}_{best}(o,p)$ to schedule operations.

Finally, the time complexity of the Minimize Start Time procedure (MSTP) is $\mathcal{O}(E)$, where E denotes the number of edges in $\mathcal{A}lg$ [1]. The time complexity of the iFTBAR algorithm is computed as follows. In each iteration of the loop, the computational complexity of the micro-steps ①, ②, ③, ④, ⑥ and ⑦ is $\mathcal{O}(PN)$, where N denotes the number of operations in $\mathcal{A}lg$ and P denotes the number of processors in $\mathcal{A}rc$. Since there are $\mathcal{N}cf$+1 ($\leq P$) calls to MSTP and since $E < N^2$, the time complexity of the micro-steps ⑤ is $\mathcal{O}(PN^2)$. Thus, for n iterations the overall time complexity of iFTBAR algorithm is $\mathcal{O}(nPN^2)$. Finally, since exactly one operation is scheduled at each iteration, $n = N$, and the time complexity is thus $\mathcal{O}(PN^3)$.

4.4 An example

We have implemented our fault tolerant heuristic iFTBAR within the SynDEx tool [22]. To illustrate the principles of our heuristic, we apply it to the example of Figure 2(a) for $\mathcal{A}lg$ and Figure 2(b) for $\mathcal{A}rc$. The execution characteristics of each `comp`/`extio` and `comm` are specifi ed by Table 1. The user requires the system to tolerate one component failures, i.e., $\mathcal{N}cf = 1$, and requires the run-time of the system to be less than 15 time units, i.e., $\mathcal{R}tc = 15.00$.

After the fi rst four steps of our heuristic, we obtain the temporary schedule of Figure 9, where two replicas of I, I', B and C are scheduled on P2 and P4.

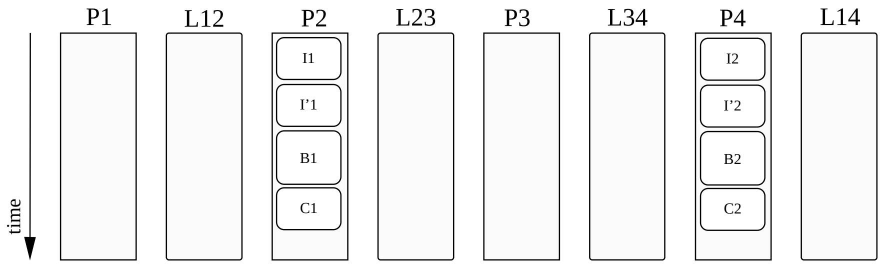

Figure 9: The temporary schedule at step 4 of the heuristics.

In the next step, $O^{(5)}_{cand} = \{$A,O'$\}$ and A is selected as the urgent operation. The two processors P1 and P3 are selected to be the optimal processors P_{opts} for A, so the two replicas of A are scheduled on P1 and P3, as shown in Figure 10. The `comms` required by these two replicas are also scheduled on the communication links L12, L23, L34, and L14.

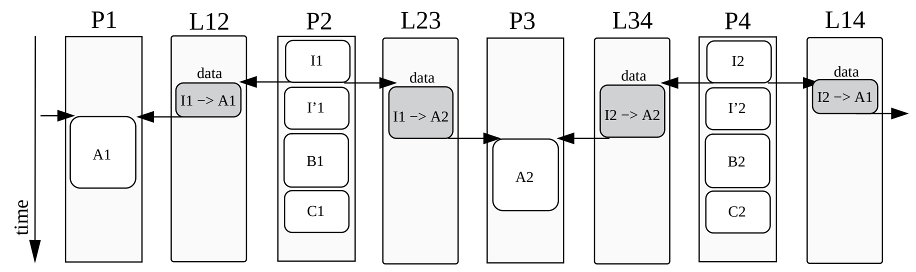

Figure 10: Schedule two replicas of A.

The start time of the two replica of A can be reduced by scheduling two additional replicas of I, the LIP of A, on P1 and P3 respectively, as shown in Figure 11. As we can see, this has the effect of suppressing all the interprocessor communications.

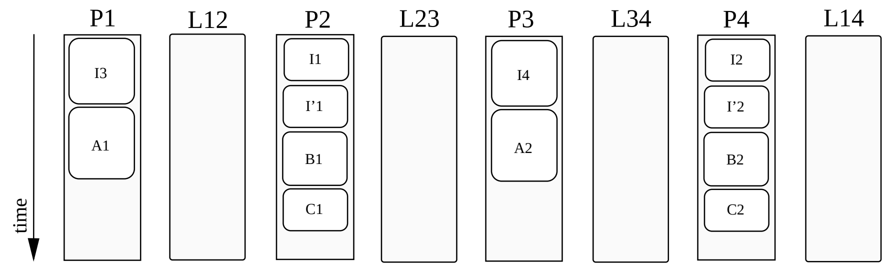

Figure 11: Minimize the start time of the replicas of A.

Then, operations D, O and O' are scheduled. At the end of our heuristic, we obtain the final schedule presented in Figure 12 (a screen capture from SynDEx). Each operation of the algorithm graph is replicated at least twice and these replicas are assigned to different processors; furthermore, each replica receives its inputs at least twice and from disjoint paths. In this example, the real-time constraint is satisfied since the total time is $13.00 < \mathcal{R}tc$.

5 Runtime Behavior

In our iFTBAR heuristics, $\mathcal{N}cf$ faults can be tolerated by scheduling $\mathcal{R}$ replicas for each operation on different processors, such that $\mathcal{R} \geq \mathcal{N}cf + 1$. If no fault occurs, each of the $\mathcal{R}$ replicas of an operation receives its inputs in parallel from the best $\mathcal{N}cf$+1 replicas of its predecessor operations in the data-flow graph; as soon as it receives the first set, the operation is executed and ignores the later $\mathcal{N}cf$ inputs. If there are k permanent faults ($k \leq \mathcal{N}cf$), each replica of an operation scheduled on a non-faulty processor receives its inputs in parallel from all the replicas of its predecessors scheduled on non-faulty processors; as soon as it receives the first set, the operation is executed and ignores the later inputs. Concerning the failure detection, there are two options:

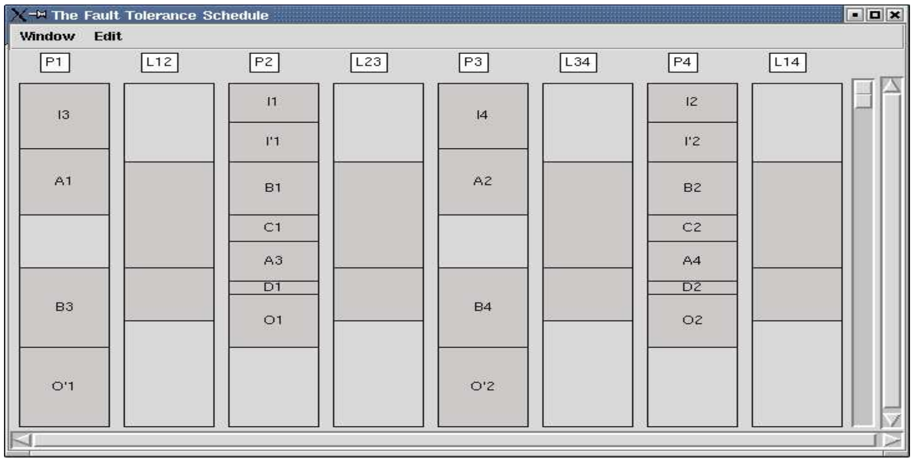

Figure 12: The final fault tolerant schedule.

1. Either we do not perform any failure detection, in which case, after a failure, the remaining components will continue to send their results to the faulty ones. This will not help in reducing the communication overheads. On the other hand, if a component experiences an *intermittent* failure, then since it will continue to receive inputs from the healthy components, it will be able to produce its results again when recovering from its intermittent failure.

2. Or we perform a failure detection procedure by knowing at what time each `comm` is supposed to happen (we are able to compute these times because the obtained schedule is *static*), and by deciding accordingly that when a `comm` did not happen, then the sending component is faulty. Each processor can therefore maintain an array of faulty components and avoid further `comms` towards these components. The drawback is that an intermittent failure cannot be recovered. Indeed, when a processor is detected to be faulty, the other healthy processors will update their array of faulty components, and will not send any more data to it. So even if this faulty component comes back to life, it will not receive any inputs and will not be able to perform any computation. Therefore, in the subsequent iterations, it will fail to send any data on its adjacent communication links, and the other healthy components will never be able to detect that it came back to life. The same reasoning applies to failure detection mistakes.

The choice between these two options can be left to the user. It will depend on the intermittent failure rate of the application as well as on the actual topology and bandwidth of the network.

6 Performance evaluation

To evaluate our fault tolerant scheduling heuristics iFTBAR, we have compared its performance with two other algorithms: HBP (Height-Based Partitioning) and NFTA (Non Fault Tolerance Algorithm); HBP is the fault tolerant algorithm proposed by Hashimoto and al. [29], which is the closest to iFTBAR that we have found in the literature, and NFTA is the modifi ed iFTBAR algorithm that produces a non fault tolerant algorithm by taking $\mathcal{N}cf = 0$. We have implemented all three algorithms within the SynDEx tool. SynDEx generates automatically executable distributed code, by fi rst producing a static distributed schedule of a given algorithm onto a given distributed architecture, and then by generating the real-time distributed executive implementing this schedule.

The performance comparisons were done in two ways, with various parameters and a variety of random $\mathcal{A}lg$ graphs: fi rst iFTBAR with $\mathcal{N}cf = 1$ against HBP with only the software redundancy of $\mathcal{A}lg$'s operation, then iFTBAR against NFTA with the software redundancy of both $\mathcal{A}lg$'s operation and communications.

The random algorithm graphs were generated as follows: given the number of operations N, we randomly generate a set of levels, each with a random number of operations. Then, operations at a given level are randomly connected to operations at a higher level to stress-test the proposed algorithm. The execution times $\mathcal{E}xe$ of each operation are randomly selected from a uniform distribution with the mean equal to the chosen average execution time. Similarly, the communication times $\mathcal{E}xe$ of each data dependency are randomly selected from a uniform distribution with the mean equal to the chosen average communication time. The average operation's execution times and data dependency's communication times are linked by the *Communication to Computation Ratio* parameter (CCR), given as an input. A CCR smaller than 1 indicates that communications are cheaper than computations, while a CCR greater than 1 indicates that communications are more expensive than computations.

6.1 Impact of duplicating operations and communications

We start our performance study by evaluating the impact of the software redundancy of both $\mathcal{A}lg$'s operation and communications on the iFTBAR algorithm. For this, we have applied iFTBAR to a set of random algorithm graphs with $N = 50$, and $CCR = 0.1, 0.5, 1.0, 5.0$. The architecture graph was a fully connected network of 6 processors (fully connected to comply with HBP's hypotheses). In this simulation experiment, both operations and communications are allowed to be replicated at least twice. Thus, we have compared the average fault tolerance schedule overheads produced by iFTBAR for $\mathcal{N}cf = 1$, averaged over 50 random $\mathcal{A}lg$ graphs. The average schedule overheads is computed in the following way:

$$Overhead = \frac{schedule\ length(iFTBAR) - schedule\ length(NFTA)}{schedule\ length(NFTA)} \times 100$$

where, $schedule\ length$(iFTBAR) (resp. $schedule\ length$(NFTA)) is the schedule length produced by iFTBAR for $\mathcal{N}cf = 1$ (resp. $\mathcal{N}cf = 0$).

We have plotted in Figure 13 the average fault tolerance overheads as a function of CCR. It shows that the average overheads decreases with CCR: this is due to the use of the locality of computations in our algorithm through the Minimize Start Time procedure.

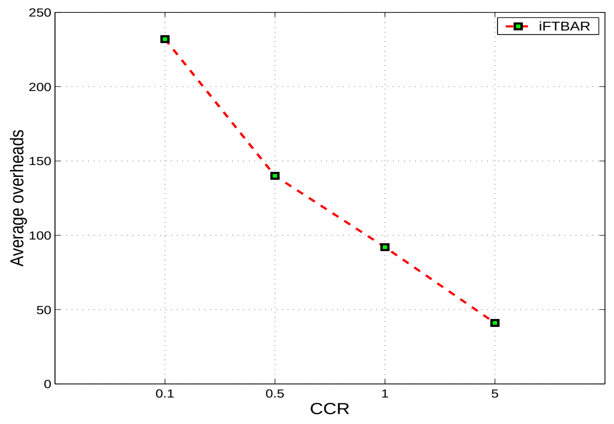

Figure 13: Impact of duplicating both operations and communications for $\mathcal{N}cf = 1$, $P = 6$ and $N = 50$.

6.2 Performance of iFTBAR against HBP

The performance measure used in this simulation is the fault tolerance overheads, computed in the following way:

$$Overhead = \frac{schedule\,length(iFTBAR\,or\,HBP) - schedule\,length(NFTA)}{schedule\,length(NFTA)} \times 100$$

where, $schedule\,length$(HBP) is the schedule length produced by HBP.

Since HBP assumes homogeneous systems and only uses software redundancy of the algorithm's operations to tolerate exactly one processor failure, iFTBAR is downgraded to these assumptions, so that the comparison be meaningful. We thus set $\mathcal{N}cf$ to one in this simulation study.

We have applied iFTBAR and HBP to a set of random algorithm graphs with a wide range of parameters: $N = 10, 20, \ldots, 80$, and $CCR = 0.1, 0.5, 1, 5, 10$. The architecture graph was a fully connected network of 4 processors. We have compared the average fault tolerance schedule overheads produced by iFTBAR and HBP, averaged over 60 random $\mathcal{A}lg$ graphs.

We have plotted in Figures 14 and 15 the average fault tolerance overheads as a function of N and CCR, both in the absence (Figures 14(a) and 15(a)) and in the presence of one arbitrary processor failure (Figures 14(b) and 15(b)); here we have computed the average overheads when each of the four processors failures, and plotted the max overheads over these four processors.

Figure 14 shows that average overheads increases with N. This is due to the active replication of all operations and communications. We also see that that iFTBAR performs better than HBP, by roughly 20%.

Figure 15 shows that the average overheads decreases while CCR increases. For $CCR \leq 1$, there is little difference between HBP and iFTBAR. In contrast, for $CCR \geq 2$, iFTBAR performs signifi cantly better than HBP (by at least 20%). This is due to our dependable schedule pressure cost function, which tries to minimize the length of the critical path.

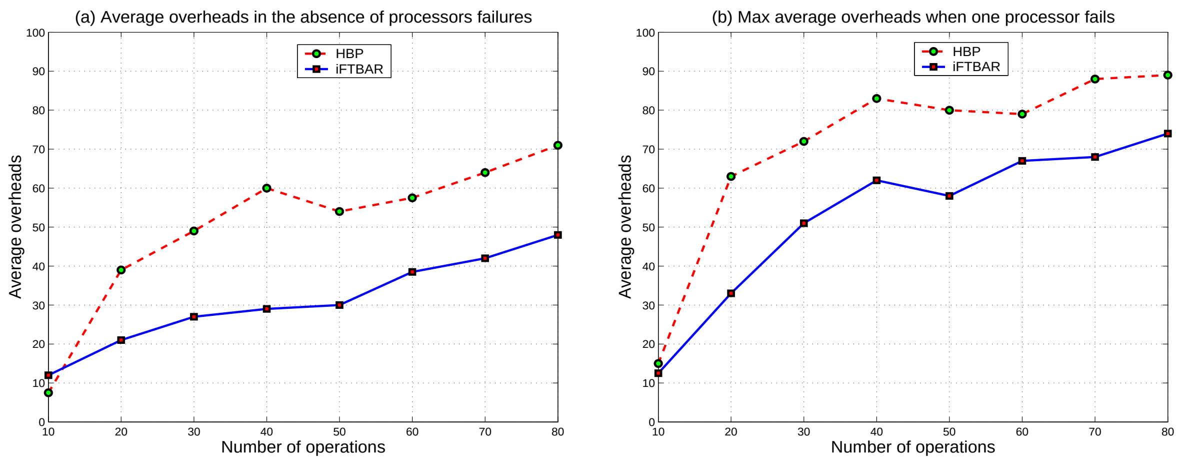

Figure 14: Impact of the number of operations for $\mathcal{N}cf = 1$, $P = 4$ and $CCR = 5$.

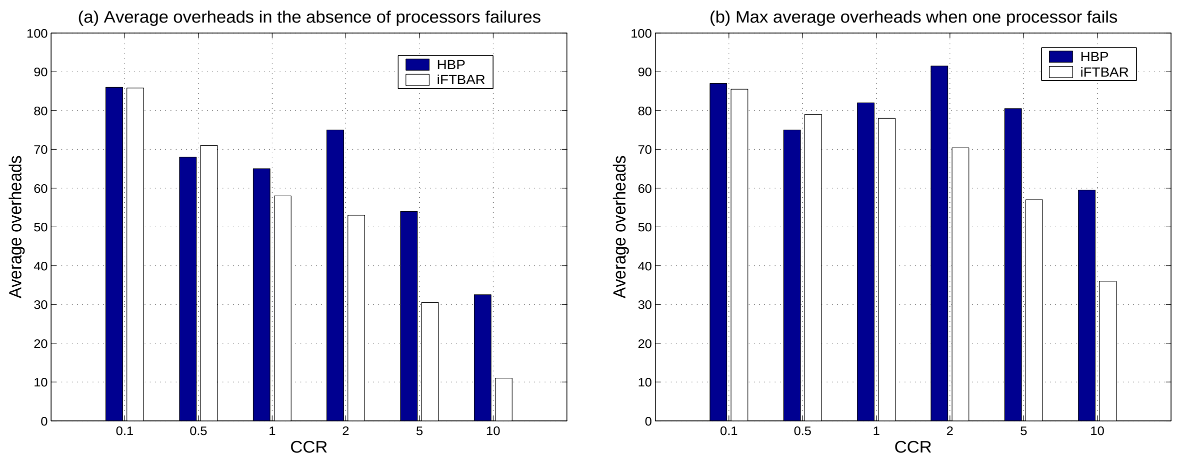

Figure 15: Impact of the communication-to-computation ratio for $\mathcal{N}cf = 1$, $P = 4$ and $N = 50$.

Finally, let us note that the time complexity of iFTBAR, $\mathcal{O}(PN^3)$ is less than the time complexity of HBP, $\mathcal{O}(PN^4)$. The reason is that HBP investigates more possibilities than iFTBAR when selecting the processor for a candidate operation [29].

7 Conclusion and future work

The literature about fault tolerance of distributed and/or embedded real-time systems is very abundant. Yet, there are few attempts to combine fault tolerance and automatic generation of distributed code for embedded systems. In this paper, we have studied this problem and proposed a software implemented fault tolerance solution.

We have proposed a new scheduling heuristics, called iFTBAR (improved Fault Tolerance Based Active Replication), which produces automatically a static distributed fault tolerant schedule of a given algorithm onto a given distributed architecture. Our solution is based on the software redundancy of both the computation operations and the communications. The produced schedules tolerate $\mathcal{N}cf$ hardware component failures, be it of processors or communication links. This is achieved by replicating all computation operations at least $\mathcal{N}cf$+1 times on distinct processors; all the best of these $\mathcal{N}cf$+1 replicated operations send their results but only the one which is received first by the destination processor is used; the other results are discarded. All the components are assumed to have a fail-silent behavior, and the network topology is assumed to have at least $\mathcal{N}cf$+1 disjoint paths between any two processors (an assumption weaker than being fully connected).

The implementation uses a scheduling heuristics for optimizing the critical path of the obtained distributed schedule. It is best suited to architectures with point-to-point links. There is some communication overheads, but on the other hand, several failures in a row can be tolerated. Also, depending on the failure detection mechanism chosen, intermittent failures can be tolerated as well.

We have implemented our iFTBAR heuristics within the SynDEx tool [22]. SynDEx is able to generate automatically executable distributed code, by first producing a static distributed schedule of a given algorithm on a given distributed architecture, and then by generating a real-time distributed executive implementing this schedule. We have also implemented the HBP (Height-Based Partitioning [29]) heuristics for comparison purpose. Although HBP only considers homogeneous architectures and only tolerates one processor failure, it is the closest to our work that we have found in the literature. The experimental results show that iFTBAR performs significantly better than HBP, both in the absence and in the presence of failures.

Finally, we are currently working on new solutions to take also the failures of sensors and actuators into account. This raises many problems: How to validate an input produced by several sensors? How to guarantee the coherence between replicated actuators? How to deal with disjoint architectures?

References

[1] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 9, pages 872–892, September 1998.

[2] K. Ahn, J. Kim, and S. Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS)*, Taipei, TAIWAN, December 1997.

[3] R. Al-Omari, Arun K. Somani, and G. Manimaran. A new fault-tolerant technique for improving the schedulability in multiprocessor real-time systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, San Francisco, April 2001. IEEE Computer Society.

[4] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental concepts in dependability. In *Proceedings of the 3rd IEEE Information Survivability Workshop (ISW-2000)*, pages 7–12, Boston, Massachusetts, USA, October 2000.

[5] A. Banerjea, C. Parris, and D. Ferrari. Recovering guaranteed performance service connections from single and multiple faults. In *Proc. GLOBECOM'94*, San Francisco, CA, November 1994.

[6] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, september 1991.

[7] A.A. Bertossi and L.V. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems Journal*, 7(3):229–245, 1994.

[8] R. Boppana and S. Chalasan. Fault-tolerant multicast communication in multicomputers. In *Proceedings of the 1995 International Conference on Parallel Processing*, volume 1, pages 118–125, 1995.

[9] A. Cherif, M. Suzuki, and T. Katayama. A novel replication technique for detecting and masking failures for parallel software: Active parallel replication. *IEICE Transactions on Information and Systems, Special Issue on Architectures, Algorithms and Networks For Massively Parallel Computing*, E80-D(9):886–892, september 1997.

[10] P. Chevochot and I. Puaut. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategie. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 356–363, HongKong, China, December 1999.

[11] M. D. Cin, W. Hohl, and V. Sieh. Hardware-supported fault tolerance for multiprocessors. In *Architecture of Computing Systems (ARCS'97)*, 1997.

[12] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 410–417, Mantova, Italy, February 2001.

[13] S. Dulman, T. Nieberg, J. Wu, and P. Havinga. Trade-off between traffic overhead and reliability in multipath routing for wireless sensor networks. In *Wireless Communications and Networking Conference*, 2003.

[14] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.

[15] P. Fragopoulou and S. G. Akl. Fault tolerant communication algorithms on the star network using disjoint paths. In *Proceedings of the 28th annual hawaii international conference on system sciences (HICSS'95)*, kingston, ontario, canada, 1995.

[16] 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.

[17] S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems*. PhD thesis, University of Pittsburgh, 1996.

[18] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedule. In *The International Conference on Dependable Systems and Networks*, San Francisco, California, USA, June 2003.

[19] A. Girault, H. Kalla, and Y. Sorel. Une heuristique d'ordonnancement et de distribution tolrante aux pannes pour systmes temps-rel embarqus. In *Modélisation des Systèmes Réactifs, MSR'03*, pages 145–160, Metz, France, October 2003. Hermes.

[20] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. In *21st International Conference on Distributed Computing Systems, ICDCS'01*, pages 695–698, Phoenix, USA, april 2001. IEEE. Extended abstract.

[21] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, FTPDS'01*, San Francisco, USA, april 2001. IEEE.

[22] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *MEMOCODE'2003, Formal Methods and Models for Codesign Conference*, Mont Saint-Michel, France, June 2003.

[23] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Proceeding Conference on Reliable Software Technologies*, pages 38–57. Springer-Verlag, 1996.

[24] K. P. Gummadi, M. J. Pradeep, and C. S. Ram Murthy. An efficient primary-segmented backup scheme for dependable real-time communication in multihop networks. *IEEE/ACM Transactions on Networking*, 11(1), Feb 2003.

[25] M. Gupta and E. Schonberg. Static analysis to reduce synchronization cost in data-parallel programs. In *23rd Symposium on Principles of Programming Languages*, pages 322–332, january 1996.

[26] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[27] S. Han and K.G. Shin. Efficient spare-resource allocation for fast restoration of real-time channels from network component failures. In *IEEE Real-Time Systems Symposium*, 1997.

[28] S. Han and K.G. Shin. Experimental evaluation of failure-detection schemes in real-time communication networks. In *IEEE Fault-Tolerant Computing Symposium*, 1998.

[29] K. Hashimoto, T. Tsuchiya, and T. Kikuno. Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. *IEICE Transactions on Information and Systems*, E85-D(3):525–534, march 2002.

[30] M. Hiller. Software fault-tolerance techniques from a real-time systems point of view. Technical report, Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Gteborg Sweden, november 1998.

[31] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[32] N. Kandasamy, J. P. Hayes, and B.T. Murray. Dependable communication synthesis for distributed embedded systems. In *22nd Int'l Conf. on Computer Safety, Reliability and Security (SAFECOMP 2003)*, Edinburgh, UK, Sept 2003.

[33] B. Kao, H. Garcia-Molina, and D. Barbara. Aggressive transmissions of short messages over redundant paths. In *IEEE Transactions on Parallel and Distributed Systems*, volume 5, pages 102–109, January 1994.

[34] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, October 2002.

[35] G. Manimaran and C. Siva Ram Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1137–1151, november 1998.

[36] Y. Oh and S. H. Son. Scheduling real-time tasks for dependability. *Journal of Operational Research Society*, 48(6):629–639, June 1997.

[37] 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, France, February 2004. IEEE.

[38] C. Pope. The scheduling of fault recovery for real-time channels. In *the Second Australasian Parallel and Real-Time Conference*, september 1995.

[39] X. Qin, Z.F. Han, H. Jin, L. P. Pang, and S. L. Li. Real-time fault-tolerant scheduling in heterogeneous distributed systems. In *Proceeding of the International Workshop on Cluster Computing-Technologies, Environments, and Applications (CC-TEA'2000)*, Las Vegas, USA, June 2000.

[40] X. Qin, H. Jiang, and D. R. Swanson. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. In *Proceedings of the 31th International Conference on Parallel Processing (ICPP 2002)*, pages 360–386, Vancouver, British Columbia, Canada, August 2002.

[41] P. Ramanathan and K.G. Shin. Delivery of time-critical messages using a multiple copy approach. *ACM Transactions on Computer Systems*, 10(2):144–166, may 1992.

[42] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2):189–219, 1994.

[43] Y. Sorel. Massively parallel computing systems with real time constraints, the algorithm architecture adequation methodology. In *the Massively Parallel Computing Systems*, may 1994.

[44] R. Sriram, G. Manimaran, and C. Siva Ram Murthy. An integrated scheme for establishing dependable real-time channels in multihop networks. In *Proc. ICCCN*, pages 528–533, 1999.

[45] N. Suri and K. Ramamritham. Editorial: Special section on dependable real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):529–531, june 1999.

[46] W. Torres-Pomales. Software fault tolerance: A tutorial, October 2000. National Aeronautics and Space Administration (NASA) Langley Research Center.

[47] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[48] Q. Zheng and K. G. Shin. Fault-tolerant real-time communication in distributed computing systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 470–480, 1998.