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 (Alg), a specification of the target distributed architecture (Arc), some distribution constraints (Dis), 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 (Exe), some real-time constraints (Rtc), and a fixed number of components failures to be tolerated (Ncf); components are processors and communication links. Our goal is to find a static schedule of Alg onto Arc, satisfying Dis, and tolerant to at most Ncf component failures, with an indication whether or not this schedule satisfies Rtc w.r.t. Exe. 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).
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 first 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, 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 fixed 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 [7], 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 $R_{tc}$ before the execution. If $R_{tc}$ is not satisfied, we can give a warning to the designer, so that he can decide whether to add more hardware or to relax $R_{tc}$.

3. The source algorithm $Alg$ can be designed with a high-level programming language based on a formal mathe-
mathematical semantics. For instance, this is the case of synchronous languages, which are moreover well suited to
the programming of embedded critical systems[23][24]. The advantage is that $A_lg$ 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 $N_{cf}$ 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[25],
a system level CAD software tool for optimizing the implementation of real-time embedded applications on multicom-
ponent architecture. Prior work has been published in[17][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 $Arc$, where each vertex is a processor, and each edge is a bidirectional com-
munication 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.

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

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

Figure 2: Example of (a) an algorithm graph $A_lg$ and (b) an architecture graph $Arc$. 
3.2 Algorithm model

The algorithm to be distributed is modeled by a data-flow graph \( Alg \). 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 \( Alg \) can be either:

- a computation operation (\texttt{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 (\texttt{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 (\texttt{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 \( A \rightarrow D \) corresponds to the sending of some arithmetic result computed by A and needed by D.

<table>
<thead>
<tr>
<th>operation</th>
<th>time</th>
<th>I</th>
<th>I'</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>O</th>
<th>O'</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2.5</td>
<td>∞</td>
<td>2.5</td>
<td>3.0</td>
<td>2.0</td>
<td>1.5</td>
<td>3.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>2.0</td>
<td>1.0</td>
<td>0.5</td>
<td>2.0</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2.5</td>
<td>∞</td>
<td>2.5</td>
<td>3.0</td>
<td>2.0</td>
<td>1.5</td>
<td>3.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>2.0</td>
<td>1.0</td>
<td>0.5</td>
<td>2.0</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>data-dependency</th>
<th>time</th>
<th>I \rightarrow A</th>
<th>I \rightarrow B</th>
<th>I' \rightarrow B</th>
<th>I' \rightarrow C</th>
<th>A \rightarrow D</th>
<th>B \rightarrow D</th>
<th>C \rightarrow O'</th>
<th>B \rightarrow O'</th>
<th>D \rightarrow O</th>
</tr>
</thead>
<tbody>
<tr>
<td>L12</td>
<td>1.0</td>
<td>2.0</td>
<td>1.5</td>
<td>2.0</td>
<td>1.5</td>
<td>2.0</td>
<td>1.5</td>
<td>1.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>L23</td>
<td>2.0</td>
<td>4.0</td>
<td>3.0</td>
<td>3.0</td>
<td>4.0</td>
<td>4.0</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>L14</td>
<td>1.0</td>
<td>2.0</td>
<td>1.5</td>
<td>2.0</td>
<td>1.5</td>
<td>2.0</td>
<td>1.5</td>
<td>1.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>L34</td>
<td>2.0</td>
<td>4.0</td>
<td>3.0</td>
<td>3.0</td>
<td>4.0</td>
<td>4.0</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Distributed constraints \( Dis \) and execution/transmission times \( Exe \) for operations and data-dependencies.

To each operation \( o \) of \( Alg \), we associate in a table \( Exe \) its execution time on each processor: each pair \( (o, p) \) of \( Exe \) 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 \( Alg \), we associate in a table \( Exe \) its transmission time on each communication link: each pair \( (d, l) \) of \( Exe \) 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 '$\infty$' to some pairs $(o, p)$ 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 specific co-processor to execute, for instance.

### 3.3 Failure model

As said in the introduction, our goal is to find an $N_{cf}$ 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 defined as a schedule in which no real-time constraints $\mathcal{R}_{tc}$ are missed, despite $N_{cf}$ 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) 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 disjoint paths, which allows at least $N_{cf} + 1$ replicas of an operation to be scheduled on different processors, which are run in parallel to tolerate at most $N_{cf}$ 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 $\mathcal{R} \geq N_{cf} + 1$. Each of the best $N_{cf} + 1$ 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 $N_{cf} + 1$ times via disjoint paths; as soon as it receives the first 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 ($N_{cf}=1$); 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.](image)

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 $I1 \rightarrow A1$ and $I2 \rightarrow A1$ are implemented via two disjoint paths, and so are $I1 \rightarrow A2$ and $I2 \rightarrow A2$.

![Figure 4: Schedule $N_{cf}+1$ replicas of each operation with $N_{cf}=1$.](image)

Since the communication cost between operations assigned to the same processor is considered to be zero, replicating an operation more than $N_{cf}+1$ 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 6 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 5.

![Schedule Diagram](image)

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

![Temporary Schedule Diagram](image)

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

### 4.2 Notations

Before describing our proposed scheduling heuristics, we first define 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.

- $F^{n}_{sched}$: 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.

- $E_{exec}(o_i, p_j)$: The end of execution time of operation $o_i$ scheduled on processor $p_j$. 
\[ E_{\text{com}}^{(n)}(o_i, o_j) : \text{The end of data communication time from operation } o_i \text{ to operation } o_j. \]

\[ S^{(n)}(o_i) : \text{The latest start time from end of } o_i, \text{ defined to be the length of the longest path from the outputs operations to } o_i. \]

\[ S_{\text{best}}^{(n)}(o_i, p_i) : \text{The earliest time at which operation } o_i \text{ can start execution on processor } p_i, \text{ computed as follows:} \]

\[
S_{\text{best}}^{(n)}(o_i, p_i) = \max_{o_j \in \text{pred}(o_i)} \left\{ N_{pf} + 1 \min_{k=1}^{N_{pf}} E_{\text{com}}^{(n)}(o_j^k, o_i) \right\}
\]

where \( o_j^k \) is the \( k \)th replica of \( o_j \). If \( o_i \) and \( o_j^k \) are scheduled in the same processor \( p_i \), then

\[
E_{\text{com}}^{(n)}(o_j^k, o_i) = E_{\text{exc}}^{(n)}(o_j^k, p_i)
\]

\[ S_{\text{worst}}^{(n)}(o_i, p_i) : \text{The latest time at which operation } o_i \text{ can start execution on processor } p_i, \text{ taking into account all the predecessors replicas; it is computed as follows:} \]

\[
S_{\text{worst}}^{(n)}(o_i, p_i) = \max_{o_j \in \text{pred}(o_i)} \left\{ N_{pf} + 1 \max_{k=1}^{N_{pf}} E_{\text{com}}^{(n)}(o_j^k, o_i) \right\}
\]

where \( o_j^k \) is the \( k \)th replica of \( o_j \). If \( o_i \) and \( o_j^k \) are scheduled in the same processor \( p_i \) then

\[
E_{\text{com}}^{(n)}(o_j^k, o_i) = E_{\text{exc}}^{(n)}(o_j^k, p_i)
\]

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 \( \tilde{\delta}^{(n)} \) of the schedule pressure \( \sigma^{(n)} \) defined in [1]. The schedule pressure \( \sigma^{(n)} \) is computed, for each operation \( o_i \in O_{\text{cand}}^{(n)} \) and each processor \( p_j \subset \mathcal{P} \), as follows:

\[
\sigma^{(n)}(o_i, p_j) := S_{\text{best}}^{(n)}(o_i, p_j) + S^{(n)}(o_i) - R^{(n-1)} \tag{1}
\]

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

\[
\tilde{\delta}^{(n)}(o_i, p_j) := S_{\text{worst}}^{(n)}(o_i, p_j) + S^{(n)}(o_i) - R^{(n-1)} \tag{2}
\]

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

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

where

\[
P_{\text{opts}}(o_i) := \left\{ p_j | p_j \in \min_{p_j \in \mathcal{P}} \tilde{\delta}^{(n)}(o_i, p_j) \right\} \tag{4}
\]

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 \[\text{Algorithm}, \text{Arc}, \text{Exec}, \text{Rtc}, \text{Dis}, \text{Ncf};\]

\[\text{Output: a fault-tolerant distributed schedule } \mathcal{FT}_{\text{sched}};\]

\begin{itemize}
  \item Initialize the lists of candidate and scheduled operations:
    \begin{equation}
    n := 0;
    \end{equation}
    \begin{equation}
    \mathcal{O}^{(0)}_{\text{cand}} := \{o \in \mathcal{O} \mid \text{pred}(o) = \emptyset\};
    \end{equation}
    \begin{equation}
    \mathcal{FT}^{(0)}_{\text{sched}} := \emptyset;
    \end{equation}
  \item \textbf{while } \mathcal{O}^{(n)}_{\text{cand}} \neq \emptyset \textbf{ do}
    \begin{enumerate}
    \item Compute the modified schedule pressure \(\hat{\delta}^{(n)}\) for each operation \(o_{\text{cand}}\) of \(\mathcal{O}^{(n)}_{\text{cand}}\) on each processor \(p_j\) of \(\mathcal{P}\) using Equation \[(\ref{eq:modified_schedule_pressure}).\]
    \item Compute the set of optimal processors \(P_{\text{opts}}\) for each candidate operation \(o_{\text{cand}}\) of \(\mathcal{O}^{(n)}_{\text{cand}}\) using Equations \[(\ref{eq:optimal_processors})\text{ and } (\ref{eq:optimal_processors_1}).\]
    \item Select the best candidate operation \(o_{\text{best}}\), such that:
    \begin{equation}
    \delta_{\text{urgent}}^{(n)}(o_{\text{best}}) := \max_{o_{\text{cand}} \in \mathcal{O}^{(n)}_{\text{cand}}} \delta^{(n)}(o_{\text{cand}}, P_{\text{opts}});
    \end{equation}
    \item Schedule \(N_{\text{cf}} + 1\) replicas of the best candidate operation \(o_{\text{best}}\) on each processor of \(P_{\text{opts}}\) computed at micro-step \(\text{step}\) \(\text{fig} 6\).
    \end{enumerate}
    \end{enumerate}
\end{itemize}

\[\text{The iFTBAR Algorithm:}\]
\[\text{Inputs: Alg, Arc, Exec, Rtc, Dis, Ncf;}\]
\[\text{Output: a fault-tolerant distributed schedule } \mathcal{FT}_{\text{sched}};\]
\[\text{begin}\]
\[\text{Initialize the lists of candidate and scheduled operations:}\]
\[n := 0;\]
\[\mathcal{O}^{(0)}_{\text{cand}} := \{o \in \mathcal{O} \mid \text{pred}(o) = \emptyset\};\]
\[\mathcal{FT}^{(0)}_{\text{sched}} := \emptyset;\]
\[\text{while } \mathcal{O}^{(n)}_{\text{cand}} \neq \emptyset \textbf{ do}\]
\[\text{1. Compute the modified schedule pressure } \hat{\delta}^{(n)} \text{ for each operation } o_{\text{cand}} \text{ of } \mathcal{O}^{(n)}_{\text{cand}} \text{ on each processor } p_j \text{ of } \mathcal{P} \text{ using Equation } (\ref{eq:modified_schedule_pressure}).\]
\[\text{2. Compute the set of optimal processors } P_{\text{opts}} \text{ for each candidate operation } o_{\text{cand}} \text{ of } \mathcal{O}^{(n)}_{\text{cand}} \text{ using Equations } (\ref{eq:optimal_processors}) \text{ and } (\ref{eq:optimal_processors_1}).\]
\[\text{3. Select the best candidate operation } o_{\text{best}}, \text{ such that:}\]
\[\delta_{\text{urgent}}^{(n)}(o_{\text{best}}) := \max_{o_{\text{cand}} \in \mathcal{O}^{(n)}_{\text{cand}}} \delta^{(n)}(o_{\text{cand}}, P_{\text{opts}});\]
\[\text{4. Schedule } N_{\text{cf}} + 1 \text{ replicas of the best candidate operation } o_{\text{best}} \text{ on each processor of } P_{\text{opts}} \text{ computed at micro-step } \text{fig} 6.\]
\[\text{The communications implied by this scheduling decision are also scheduled here, such that each replica of } o_{\text{best}} \text{ receives its input data via disjoint links from only the best } N_{\text{cf}} + 1 \text{ replicas of each of these predecessor operations.}\]
\[\text{5. Try to reduce the worst start time } S_{\text{worst}}^{(n)} \text{ of each replica of } o_{\text{best}} \text{ by applying the procedure “Minimize Start Time” (see Fig. } \text{fig} 7).\]
\[\text{6. Update the lists of scheduled and candidate operations:}\]
\[\mathcal{FT}^{(n+1)}_{\text{sched}} := \mathcal{FT}^{(n)}_{\text{sched}} \cup \{o_{\text{best}}, P_{\text{opts}}(k), S_{\text{best}}(o_{\text{best}}, P_{\text{opts}}(k))\};\]
\[\mathcal{O}^{(n+1)}_{\text{cand}} := \mathcal{O}^{(n)}_{\text{cand}} \cup \{o' \in \text{succ}(o_{\text{best}}) \mid \text{pred}(o') \subseteq \mathcal{FT}^{(n+1)}_{\text{sched}}\};\]
\[n := n + 1;\]
\[\text{end while}\]
\[\text{return the fault-tolerant distributed schedule } \mathcal{FT}_{\text{sched}};\]
\[\text{end}\]

Figure 7: The fault tolerant scheduling algorithm iFTBAR.

\[\text{FTBAR was first presented in } \cite{[\text{18}],[\text{19}]}. \text{ The difference with iFTBAR is that it tolerates only processor failures.}\]
At each step \( n \), one operation of the list \( O^{(n)}_{\text{cand}} \) is selected to be scheduled. To select an operation, we select at the micro-steps ➀ and ➁ for each candidate operation \( o_{\text{cand}} \), the \( N cf+1 \) optimal processors \( P_{\text{opts}} \) having the minimum schedule pressure. Then, among those pairs \( \langle o_{\text{cand}}, P_{\text{opts}} \rangle \), we select at the micro-step ➂ the best one \( \langle o_{\text{best}}, P_{\text{opts}} \rangle \) having the maximum schedule pressure, i.e., the most urgent pair.

The selected operation \( o_{\text{best}} \) is replicated and scheduled at the micro-step ➃ on each processor of \( P_{\text{opts}}(o_{\text{best}}) \), and the \( \text{comms} \) 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 \( o_{\text{best}} \) is possibly reduced by replicating its predecessors using the procedure \textit{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 \( p \), \( S_{\text{worst}} \), a schedule \( FT_{\text{sched}} \);

Output: an improved schedule \( FT_{\text{sched}} \);

begin

➊ if (\( o \) cannot be scheduled on \( p \)) or (\( o \) is an input operation) then return \( FT_{\text{sched}} \);

➋ Find out the Latest Immediate Predecessor (LIP) of \( o \);

➌ Minimize the start time of this LIP by recursively calling this procedure:

\[
FT_{\text{sched}} := \text{MSTP}(\text{LIP}, p, S_{\text{worst}}(\text{LIP}, p), FT_{\text{sched}});
\]

➍ Compute the new \( S_{\text{worst}}(o, p) \);

❼ if (new \( S_{\text{worst}}(o, p) \geq S_{\text{worst}}(o, p) \)) then

• Undo all the replications just performed in ➌;

• Reschedule \( o \) to \( p \) at \( S_{\text{best}}(o, p) \); the \( \text{comms} \) implied by this schedule are also scheduled here.

• return \( FT_{\text{sched}} \);

⪌ else

• \( FT_{\text{sched}} := FT_{\text{sched}} \cup \{ \text{LIP}, p, S_{\text{best}}(\text{LIP}, p) \} \);

• Find out the new LIP of \( o \) and goto ➋.

end

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

For each replica \( o_{\text{best}}^k \) of the selected candidate operation \( o_{\text{best}} \) and for each pair \( \langle o_j, o_{\text{best}}^k \rangle \) where \( o_j \in \text{pred}(o_{\text{best}}) \), the data dependency \( \langle o_j, o_{\text{best}}^k \rangle \) is implemented as \( N cf+1 \) \( \text{comms} \) assigned to as many disjoint paths. However, if there exists one replica of \( o_j \) implemented on the same processor as \( o_{\text{best}}^k \), then no \( \text{comms} \) are added since the data dependency is an intra-processor communication (see Section ⃣ 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 \(S_{worst}^{(n)}(o, p)\) to give priority to operations and \(S_{best}^{(n)}(o, p)\) to schedule operations.

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

### 4.4 An example

We have implemented our fault tolerant heuristic iFTBAR within the SYNDEX tool \([23]\). To illustrate the principles of our heuristic, we apply it to the example of Figure 2(a) for \(A_{lg}\) and Figure 2(b) for \(A_{rc}\). The execution characteristics of each comp/extio and comm are specified by Table 1. The user requires the system to tolerate one component failures, i.e., \(N_{cf} = 1\), and requires the run-time of the system to be less than 15 time units, i.e., \(R_{tc} = 15.00\).

After the first 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.](image)

In the next step, \(O_{cand}^{(5)} = \{A, O’\}\) 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 [11]. 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 $13.00 < Rtc$.

5 Runtime Behavior

In our iFTBAR heuristics, $Ncf$ faults can be tolerated by scheduling $R$ replicas for each operation on different processors, such that $R \geq Ncf + 1$. If no fault occurs, each of the $R$ replicas of an operation receives its inputs in parallel from the best $Ncf + 1$ 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 $Ncf$ inputs. If there are $k$ permanent faults ($k \leq Ncf$), 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:
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. [11], which is the closest to iFTBAR that we have found in the literature, and NFTA is the modified iFTBAR algorithm that produces a non fault tolerant algorithm by taking $N_{cf} = 0$. We have implemented all three algorithms within the SYNDEX tool. SYNDEX generates automatically executable distributed code, by first 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 $Alg$ graphs: first iFTBAR with $N_{cf} = 1$ against HBP with only the software redundancy of $Alg$‘s operation, then iFTBAR against NFTA with the software redundancy of both $Alg$‘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 $\xi_{exe}$ of each operation are randomly selected from a uniform distribution with the mean equal to the chosen average execution time. Similarly, the communication times $\xi_{exe}$ 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 $Alg$‘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 $N_{cf} = 1$, averaged over 50 random $Alg$ 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 $N_{cf} = 1$ (resp. $N_{cf} = 0$).

We have plotted in Figure [14] 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.
6.2 Performance of iFTBAR against HBP

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

\[
\text{Overhead} = \frac{\text{schedule length}(\text{iFTBAR or HBP}) - \text{schedule length}(\text{NFTA})}{\text{schedule length}(\text{NFTA})} \times 100
\]

where, \(\text{schedule length}(\text{HBP})\) 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 \(N_{cf}\) 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 \(Alg\) 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 significantly 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.
Finally, let us note that the time complexity of iFTBAR, $O(PN^3)$ is less than the time complexity of HBP, $O(PN^4)$. The reason is that HBP investigates more possibilities than iFTBAR when selecting the processor for a candidate operation $\mathcal{O}$. 

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 $N_{cf}$ hardware component failures, be it of processors or communication links. This is achieved by replicating all computation operations at least $N_{cf}+1$ times on distinct processors; all the best of these $N_{cf}+1$ 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 $N_{cf}+1$ 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


