Generation of Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems with Multi-Point Links *

Alain Girault † Christophe Lavarenne ‡ Mihaela Sighireanu § Yves Sorel ¶

Abstract

We describe a solution to automatically produce distributed and fault-tolerant code for real-time distributed embedded systems. The failures supported are processor failures, with fail-stop behavior. Our solution is grafted on the "Algorithm Architecture Adequation" method (AAA), used to obtain automatically distributed code. The heart of AAA is a scheduling heuristic that produces automatically a static distributed schedule of a given algorithm onto a given distributed architecture. We design a new heuristic in order to obtain a static, distributed and fault-tolerant schedule. The new heuristic schedules K supplementary replicas for each computation operation of the algorithm to be distributed and the corresponding communications, where K is the number of processor failures intended to be supported. In the same time, the heuristic statically computes the main replica after each failure, such that the execution time is minimized. The analysis of this heuristic shows that it gives better results for distributed architectures using multi-point, reliable links. This solution corresponds to a software implemented fault-tolerance, by mean of software redundancy of algorithm's operations and timing redundancy of communications.

Keywords: *Real-time embedded systems, multi-component architectures, software implemented fault-tolerance, Algorithm Architecture Adequation method, static scheduling, distribution heuristics.*

1 Introduction

Embedded Systems. Embedded systems account for a major part of critical applications (space, aeronautics, nuclear . . .) as well as public domain applications (automotive, consumer electronics . . .). Their main features are:

- *duality automatic-control/discrete-event*: they include control laws modeled as differential equations in sampled time and discrete event systems to schedule the control laws;
- *critical real-time*: timing constraints which are not met may involve a system failure leading to a human, ecological, and/or financial disaster;
- *limited resources*: they rely on limited computing power and memory because of weight, encumbrance, energy consumption (e.g., autonomous vehicles), radiation resistance (e.g., nuclear or space), or price constraints (e.g., consumer electronics);
- *distributed and heterogeneous architecture*: they are often distributed to provide enough computing power and to keep sensors and actuators close to the computing sites.

Synchronous Programming. Synchronous programming [17] offers specification methods and formal verification tools that give satisfying answers to the above mentioned needs. The three main synchronous languages are ESTEREL [5], LUSTRE [18], and SIGNAL [24]. These specification methods are now successfully applied in industry. For instance, LUSTRE is used to develop the control software for nuclear plants and AIRBUS planes [3]. ESTEREL is used to develop DSP chips for mobile phones [2], to design and verify DVD chips, and to program the flight control software of RAFALE fighters [4]. And SIGNAL is used to develop airplane engines. The key advantage pointed by these companies is that the synchronous approach has a rigorous mathematical semantics which allows the programmers to develop critical software faster and better.

Synchronous languages are based upon the modeling of the system with finite state automata, the specification with formally defined high level languages, and the theoretical analysis of the models to obtain formal validation methods [25, 8]. However, the following aspects, extremely important w.r.t. the target fields, are not taken into account:

- *Distribution*: Synchronous languages are parallel, but the parallelism used in the language aims only at making the designer's task easier, and is not related to the system's parallelism. Synchronous languages compilers produce centralized sequential code.

*This work was funded by INRIA under the TOLÈRE research action and has been done while Mihaela Sighireanu had a post-doctoral position at INRIA. Published in the *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, San Francisco, USA, April 2001.

†INRIA-BIP, 655 av. de l'Europe, 38330 Montbonnot, France. Tel: +33 476 61 53 51. Fax: +33 476 61 52 52. Email: Alain.Girault@inrialpes.fr.

‡INRIA-SOSSO, Christophe.Lavarenne@inria.fr

§University of Paris 7, LIAFA, Tel: +33 144 27 28 39. Email: sighirea@liafa.jussieu.fr

¶INRIA-SOSSO, Domaine de Voluceau, 78, Rocquencourt, France. Tel: +33 139 63 52 60. Email: Yves.Sorel@inria.fr

- *Fault-tolerance*: An embedded system being intrinsically critical [22, 26], it is essential to insure that its software is fault-tolerant. This can even motivate its distribution itself. In such a case, at the very least, the loss of one computing site must not lead to the loss of the whole application.

The "Algorithm Architecture Adequation" Method. The "Algorithm/Architecture Adequation" method [15, 27] (AAA for short) has been successfully used to obtain distributed code optimizing the global computing time on the given hardware. The typical target architectures are *multi-component* ones. Such architectures are built from different types of programmed components (RISC, CISC, DSP processors . . .) and/or of non-programmed components (ASIC, FPGA, full-custom integrated circuits . . .), all together connected through a network of different types of communication components (point-to-point serial or parallel links, multi-point shared serial or parallel buses, with or without memory capacity . . .). They typically include less than 10 processors.

One advantage of AAA is that it preserves the above mentioned properties of synchronous programs. Concretely, AAA takes as inputs an algorithm and an architecture specifications, along with distribution constraints and real-time constraints. It then proceeds in two steps:

1. First it produces a *static distributed schedule* of the algorithm's operations onto the processors, and of the algorithm's data-dependencies onto the communication links. The real-time performances of the implementation are optimized by taking into account interprocessor communications which are critical. This is an optimization problem and, as other resource allocation optimization problems, it is known to be NP-hard. Several heuristics have been proposed in [15, 29].

2. Then, from this static schedule, it produces automatically a *real-time distributed executive*, and ensures the synchronization between the processors, as they are required by the algorithm specification. The obtained distributed executive is guaranteed to satisfy the real-time constraints, without deadlock and with minimum overhead.

The SYNDEX [23] tool[1] implements AAA. The architecture and the algorithm can both be specified with SYNDEX's graphical user interface. The algorithm can also be imported from a file which is the result of the compilation of a source program written in synchronous languages like ESTEREL [5], LUSTRE [18], or SIGNAL [24], through the common format DC [28].

[1] SYNDEX (Synchronized Distributed Executive) is available at the url `http://www-rocq.inria.fr/syndex`

Motivation of this Work. Our goal is to produce automatically *distributed fault-tolerant code*. Taking advantage of AAA, we propose a new scheduling heuristic that will produce automatically a static distributed fault-tolerant schedule of the given algorithm onto the given distributed architecture.

Our solution must adapt existing work in fault-tolerance for distributed and real-time systems to the specificities of embedded systems and of AAA. In particular, the fault-tolerance should be obtained without any help from the user (automatically distributed constraint) or any added hardware redundancy (embedded system constraint). It will therefore fall in the class of software implemented fault-tolerance. The second requirement is essential: it implies that we have to do with the existing parallelism of the given architecture, and that we won't add extra hardware. Moreover, in order to perform optimizations and to minimize the executive overhead, the scheduling used in AAA is completely static (all scheduling decisions are taken off-line [15]), and based on the characteristics of each algorithm's operation relatively to the hardware component on which it is executed. Finally, neither the algorithm to be executed nor the architecture of the system are fixed, but they are inputs of the method. For these reasons, we cannot apply the existing methods, proposed for example in [1, 10, 6, 13, 12], which use preemptive scheduling or approximation methods.

Related Work. There exists very little work on this precise topic. Some researchers make hard assumptions on the failure models (e.g., only fail-silent, only processor failures) and on the kind of schedule desired (e.g., only static schedule). By sticking to these assumptions however, they are able to obtain automatically distributed fault-tolerant schedules (see for instance [9, 7, 20]). Other researchers take into account much less restrictive assumptions, but they only achieve hand-made solutions, e.g., with specific communication protocols, voting mechanisms, . . . (see the vast literature on fault-tolerance in distributed systems, for instance [19]).

Like the other researchers belonging to the first group, we propose an automatic solution to the fault-tolerance distributed problem. Here are the original points:

- Firstly, we design our source algorithm with a programming language based on a formal mathematical semantics (see above). The advantage is that our algorithm can be formally verified with model-checking and theorem proving tools [25, 8], and therefore we can assume that it is clear of design faults.

- Secondly, we take into account the execution duration of both the operations and the data communications to optimize the critical path of the obtained schedule.

- Thirdly, 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 before the execution. If the real-time constraints are not satisfied, we can give a warning to the designer, so that he can decide to add more hardware or to relax his real-time constraints.

Paper Outline. Section 2 states our fault-tolerance problem, and presents the various models used by AAA. Section 3 presents the proposed solution for providing fault-tolerance within AAA. Finally, Section 4 summarizes the more important issues and gives some concluding remarks.

2 Fault-Tolerance Problem and AAA Models

Fault-Tolerance Problem. Given an algorithm specified as a data-flow graph, a distributed architecture specified as a graph, some distribution constraints, some real-time constraints, and a number K, produce automatically a distributed schedule for the algorithm onto the architecture w.r.t. the distribution constraints, satisfying the real-time constraints, and tolerant to K permanent fail-silent processor failures, by means of error compensation, using software and/or time redundancy.

Algorithm Model. The algorithm is modeled by a *data-flow graph*. Each vertex is an *operation* and each edge is a *data-flow channel*. The algorithm is executed repeatedly for each input event from the sensors in order to compute the output events for actuators. We call each execution of the data-flow graph an *iteration*. This model exhibits the potential parallelism of the algorithm through the partial order associated to the graph. Graph operations are of three kinds:

1. A computation operation (`comp`): its inputs must precede its outputs, whose values depend only on input values; there is no internal state variable and no other side effect.
2. A memory operation (`mem`): the data is held by a `mem` in sequential order between iterations; the output precedes the input, like a register in Boolean circuits.
3. An external input/output operation (`extio`): operations with no predecessor in the data flow graph (resp. no successor) stand for the external input interface (resp. output) handling the events produced by the sensors (resp. actuators). The `extios` are the only operations with side effects, however, we assume that two executions of a given input `extio` in the same iteration always produce the same output value.

Figure 1 is an example of algorithm graph, with six operations: I and O are `extios` (resp. input and output), while A–E are `comps`.

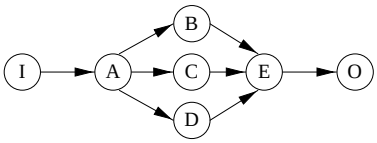

Figure 1. Example of an algorithm graph: I and O are `extios`, A–E are `comps`.

Architecture Model. The architecture is modeled by a graph, where each vertex is a processor, and each edge is a 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`, between operations allocated to different processors. Figure 2 is an example of architecture graph, with three processors and one multi-point link (i.e., a bus).

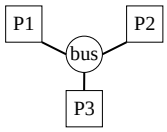

Figure 2. Example of an architecture graph with three processors and a bus.

Distribution Constraints. The distribution constraints consist in assigning to each pair ⟨operation, processor⟩ the value of the execution duration of this operation onto this processor. Each value is expressed in time units, and the value "∞" means that this operation cannot be executed on this processor. Since we also want to take into account inter-processor communications, we assign a communication duration to each pair ⟨data dependency, communication link⟩, also in time units.

For instance, the distribution constraints for the algorithmic graph of Figure 1 and the architecture graph of Figure 2 are given by the two following tables of time units:

		operation						
		I	A	B	C	D	E	O
proc.	P1	1	2	3	2	3	1	1.5
	P2	1	2	1.5	3	1	1	1.5
	P3	∞	2	1.5	1	1	1	∞

data-dependency							
I▷A	A▷B	A▷C	A▷D	B▷E	C▷E	D▷E	E▷O
1.25	0.5	0.5	1	0.5	0.6	0.8	1

Here it takes more time to communicate the data-dependency I▷A than A▷B simply because there are more data to transmit.

Implementation Model. The implementation within AAA consists in reducing the potential parallelism of the algorithm graph into the available parallelism of the architecture graph. This is formalized in terms of three graphs transformations:

1. Each `comp`/`mem`/`extio` is assigned to the computation unit of one processor according to the distribution constraints. Each inter-processor data-dependency is transformed into a vertex, called `comm`, linked to the source operation (resp. destination) with an input edge (resp. output).
2. Each `comm` generated by the first transformation is assigned to the set of communication units which are bound to the link connecting the processors executing the source and destination operations. They cooperate to transfer data between the local memories of their respective processors.
3. The `comps`/`mems`/`extios` (resp. `comms`) which have been assigned to a computation unit (resp. communication unit) during the first transformation (resp. second) are scheduled. Each schedule is completely static.

The `comms` are thus totally ordered over each communication link. 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.

Together, our models allow the specification of a broad range of systems. Indeed, a `comp` can be a single instruction (i.e., fine grain parallelism) or a function for instance written in C (i.e., coarse grain parallelism).

3 The Proposed Solution

The solution we propose consists of a new scheduling heuristic to be used in the SYNDEX tool. Its performances will be evaluated according to the following criteria:

1. The computation and communication overhead introduced by fault-tolerance.
2. The timing performances of the faulty system, i.e., a system presenting at least one failure. We distinguish the iteration in which the failure(s) actually occurs and the subsequent iterations where one or more processors are faulty but no new failure occurs. We call an iteration in which at least one failure occurs a *transient iteration*.
3. The capability to support several failures within the same iteration.
4. And finally the appropriateness to different kinds of architecture.

Principle. The proposed solution uses the software redundancy of `comps`/`mems`/`extios` and the time redundancy of `comms`. Each operation o of the algorithm graph is replicated on $K+1$ different processors of the architecture graph, where K is the number of permanent failures to be supported. Among these $K+1$ replicas, the one whose completion date is the earliest, is designated to be the *main replica*. Without entering into details, completion dates are computed according to the execution duration of each operation and each data-dependency given by the user in the distribution constraints. The main replica sends its results to each processor executing one replica of each successor operation of o, except the processors already executing another replica of o (in which case it is an intra-processor communication). The processor executing the main replica is called the *main processor* of o. The remaining K processors executing o, called *backup processors*, execute o and watch on the response of the main processor. If the main processor does not respond on time, it is considered as faulty, another main processor executing a replica of o sends o's results to the successor operations.

This solution raises the following problems:

1. *What kind of communication mechanism should be used to send results to the successors?* We choose the send/receive mechanism, where the main processor of operation o sends the results of o to all the processors executing a (main or backup) successor operation of o, and to all the backup processors of o. This mechanism is already implemented in SYNDEX for non fault-tolerant code.
2. *When is the main processor of an operation declared faulty?* With a single multi-point link (e.g., a bus), the main processor of operation o broadcasts the outputs of o while the backup processors observe the activity to detect the failure of the main processor. With point-to-point links, the detection of the main processor's failure is similar to a Byzantine agreement problem [21]. To deal with point-to-point links and to avoid heavy agreement algorithms, we have proposed in [14] another solution, based on the active redundancy on both `comps` and `comms`. In this solution, each operation is replicated $K+1$ times and each replica sends its results to each replica of each successor operation. The idea is that each operation waits until it receives its first set of inputs and discards the further inputs. There is no main replica to choose and no timeout to compute, but on the other hand, there is more communication overhead.
3. *How are computed the timeouts associated to the communications?* We choose to compute a given timeout as the worst case upper-bound of the message transmission delay. This upper-bound is computed from

the characteristics of the communication network (see Section 2). This is the least possible value avoiding multiple sendings of messages.

4. *According to which criterion is the main processor selected?* This criterion must be applied initially and each time the backup processors elect a new main processor following a failure. We choose the processor which finishes first the execution of the replica operation. For each operation, we thus compute from the static schedule a total order of all the backup processors. This total order is known by each processor, so the result of the election is the same for everybody.

Scheduling Heuristic. We present the algorithm of the heuristic implementing this solution. It is a greedy list scheduling, adapted from the non fault-tolerance heuristic presented in [15, 29].

S0. Initialize the lists of candidate and scheduled operations:
$O_{sched}(0) = \emptyset, \quad O_{cand}(0) = \{o \in O \mid pred(o) \subseteq O_{sched}(0)\}$
Sn. **while** $O_{cand}(n) \neq \emptyset$ **do**
mSn.1 Compute the scheduling pressure for each $o_i \in O_{cand}(n)$ and keep the first $K+1$ results for each operation:
$\cup_{l=1}^{l=K+1}\{\sigma^{opt}(n)(o_i, p_{i_l})\} = \min_{p_j \in P}^{K+1}\{\sigma(n)(o_i, p_j)\}$
$P^{(K+1)}(o_i) = \cup_{l=1}^{l=K+1}\{p_{i_l}\}$
mSn.2 Select the best candidate operation o such that:
$\sigma^{best}(n)(o) = \max_{o_i \in O_{cand}(n)} \cup_{l=1}^{l=K+1}\{\sigma^{opt}(n)(o_i, p_{i_l})\}$
mSn.3 Implement the operation o selected at mSn.2 on the first $K+1$ processors computed at mSn.1, as well as the implied `comms`.
The main processor is $p_m \in P^{(K+1)}(o)$ such that:
$S(n)(o, p_m) + \Delta(o, p_m) = \min_{p_l \in P^{(K+1)}(o)}\{(S(n)(o, p_l) + \Delta(o, p_l))\}$
mSn.4 Update the lists of candidate and scheduled operations:
$O_{sched}(n) = O_{sched}(n-1) \cup \{o\}$
$O_{cand}(n+1) = O_{cand}(n) - \{o\} \cup \{o' \in succ(o) \mid pred(o') \subseteq O_{sched}(n)\}$
end while

At each step $n \geq 1$, $O_{sched}(n)$ is the list of already scheduled operations, and $O_{cand}(n)$ is the list of candidate operations built from the algorithm graph. An operation is candidate if all its predecessors are already scheduled. Initially, $O_{sched}(0)$ is empty. By using a cost function called *schedule pressure*, one operation of $O_{cand}(n)$ is selected to be scheduled at step n.

The schedule pressure σ is computed in two phases. The first one is done before the scheduling heuristics. It computes, using the algorithm graph and the characteristics lookup table, the critical path of the algorithm (noted R) and, for each operation o_i the maximal date at which o_i may end (noted $E(o_i)$) computed from the end of the critical path. The second phase takes place at each step of the scheduling algorithm. It computes for an operation $o_i \in O_{cand}(n)$ and a processor $p_j \in P$ (P is the processor's set) the "earliest start date from start" (noted $S(n)(o_i, p_j)$), i.e., the execution time of the part of the distributed algorithm scheduled at the step $n-1$. $S(n)(o_i, p_j)$ takes into account the communication times between o_i and the main processor of its predecessors and successors, when they differ from p_j. This choice improves the execution time for the system without failures, but may give longer execution times in the faulty cases. Thus, σ is computed as follows:

$$\sigma(n)(o_i, p_j) = S(n)(o_i, p_j) + \Delta(o_i, p_j) + E(o_i) - R$$

where $\Delta(o_i, p_j)$ is the execution duration of o_i on processor p_j; this value is given in p_j's characteristics lookup table. The schedule pressure measures how much the scheduling of the operation lengthens the critical path of the algorithm. Therefore it introduces a priority between the operations to be scheduled.

The selected operation is obtained as follows. First, in the micro-step mSn.1, we compute for each candidate operation o_i the set $P^{(K+1)}(o_i)$ of the first $K+1$ execution units minimizing the schedule pressure. The first $K+1$ minimal schedule pressures for o_i, called $\sigma^{opt}(n)(o_i, p_{i_l})$, give the processors p_{i_l} from which the set $P^{(K+1)}(o_i)$ is computed (the superscript $(K+1)$ for P indicates its cardinality). We thus obtain for each operation $K+1$ pairs $\langle$operation, processor$\rangle$. Then, in the micro-step mSn.2, the operation belonging to the couple having the greatest schedule pressure is selected. If there exists more than one couple having the greatest schedule pressure, one is randomly chosen among them.

The implementation of the selected operation at the micro-step mSn.3 implies the choice of a main processor for the operation and the computation of timeouts for the communication operations implemented on the backup processors. We select as main processor the processor of the set $P^{(K+1)}(o)$ (the first $K+1$ processors computed for o at the micro-step mSn.1) which finishes first the execution of the operation, i.e., the one which minimizes the sum $S(n)(o, p_l) + \Delta(o, p_l)$. The K backup processors are ordered according to the increasing order of the sum $S(n)(o, p_l) + \Delta(o, p_l)$, i.e., to the increasing order of the completion date of the operation o.

Communication Overhead due to Fault-Tolerance. Now let us study the optimality of the number of inter-processor communications generated in the fault-tolerant schedule by our heuristic. Firstly, each operation of the algorithm graph is replicated $K+1$ times, but each replica only receives its inputs only once, namely from the main replica of the predecessor operation. Therefore each data-dependency of the algorithm graph leads to at most $K+1$ inter-processor communications. Indeed, when the two operations linked by the data-dependency are scheduled on the same processor, we have an intra-processor communication. In this sense we say that the number of messages in the fault-tolerant schedule is minimal.

Secondly, when a failure occurs, we claim that the number of inter-processor communications in the resulting

schedule is less than in the initial schedule. Remember that only the main replicas send their results through inter-processor communications. Let us call p the faulty processor. Consider an operation o with n successor operations in the algorithm graph, and whose main replica is assigned to p. In the initial schedule, this main replica of o sends its results to $(K+1) \times n$ replicas. Among these, a number k_{intra} are intra-processor communications because the corresponding replica is also assigned to p. The number k_{inter} of inter-processor communications actually sent by the main replica of o is such that:

$$k_{inter} + k_{intra} = (K+1) \times n$$

Now since o's main replica is assigned to p which fails, a new main processor will be chosen for o. The previous k_{intra} messages are no longer necessary since they concern operations assigned to p which is faulty. Among the remaining k_{inter} messages, some more are intra-processor because they concern operations assigned to the new main processor of o. As a result, the new number of inter-processor communications needed to send the results of o to all the replicas of all its successor operations is less than in the initial schedule.

An Example. We apply our heuristic to the example of Figures 1 and 2. The user requires to tolerate one permanent processor failure. The execution characteristics of each `comp`/`mem`/`extio` and `comm` are specified by the two tables of time units given in Section 2.

After the first two steps of our heuristic, we get the temporary schedule of Figure 3. In this diagram, an operation is represented by a white box, whose height is proportional to its execution time. Each main operation is represented by a thick white box. A `comm` is represented by a gray box, whose height is proportional to communication time, and whose ends are bound by arrows from the source operation to the destination operation.

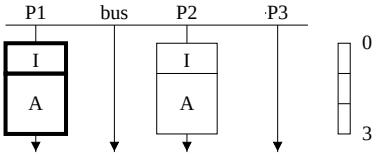

Figure 3. Temporary schedule: only operations I and A are scheduled.

At the next step, operation B is scheduled. Assigning B to P1 would save an inter-processor communication, but because of the cost of execution B on P1, the expected completion date of B would be 6. In contrast, assigning B to P2 gives an expected completion date of 4.5; therefore P2 is chosen as the main processor of B. Similarly, the expected completion date is 5 if P3 is chosen; therefore P3 is chosen as the backup processor. We obtain:

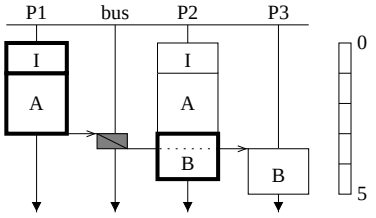

Figure 4. Temporary schedule: operations I, A, and B are scheduled.

At the end of our heuristic, we obtain the final schedule presented in Figure 5. Each operation of the algorithm graph is replicated twice and these replicas are assigned to different processors.

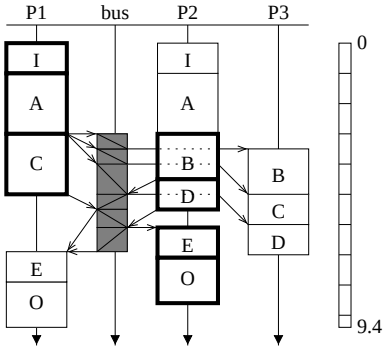

Figure 5. Final fault-tolerant schedule.

The next timing diagram shows the execution when P2 crashes: as expected, the number of communications does not increase, and the response time is increased by the waiting delay of the response from the faulty processor.

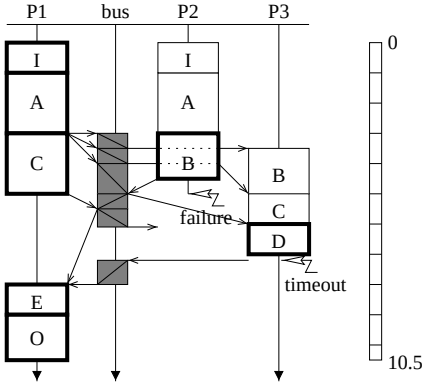

Figure 6. Timed execution when P2 crashes.

Analysis of the Example. To evaluate the overhead introduced by the fault-tolerance, let us consider the non fault-tolerant schedule produced for our example with the basic heuristic of SYNDEX. This schedule is shown in Figure 7.

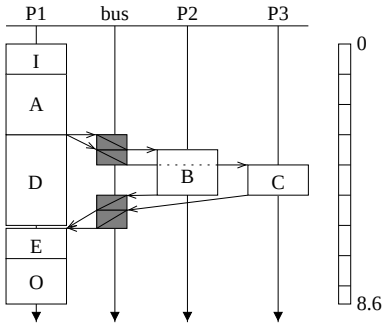

Figure 7. Non fault-tolerant schedule.

In this particular case, the overhead is therefore $9.4 - 8.6 = 0.8$. With a bigger example, the overhead would probably be larger. Part of this overhead is due to the extra computations (for the replica operations), and part is due to the extra communications (for sending their input data to all the replica operations instead of only one successor operation). Since the replica operations do not send their result until a failure occurs, the communication overhead is minimal. On the other hand, the computation overhead increases with the number K of failures to be tolerated.

When a failure occurs, extra communication can take place. This is the case of our example when P2 crashes (see Figure 6). The response time of the faulty system is greater than in absence of failures, since some overtime is necessary to detect the failure of the main processors. For the same reason, the arrival of several failures during the same iteration is not well supported since there is a risk that the sum of timeouts amassed overtakes other timeouts. As already said, the current solution is easier and cheaper to implement for architectures where the communication units are connected to a unique multi-point link. With point-to-point links, the solution presented in [14] should be preferred. It should be noted that this other solution also supports several failure occurrences during the same iteration.

If the given target architecture uses a single multi-point link (e.g., a bus), then the outputs of all operations will be broadcasted over this multi-point link to all the processors, including the faulty ones. Then a processor that was previously marked as being faulty and that is now running (because of an intermittent failure) can resume its computations and output its results on the bus. Therefore, if after a failure detection the healthy processors continue to scan the bus, they will detect that the faulty processor is now running and they will update their array of faulty processors accordingly. This scheme allows us to treat intermittent fail-silent behaviors [11].

4 Conclusion

The literature about fault-tolerance of distributed and/or embedded real-time systems is very rich. Yet, there are few attempts to combine fault-tolerance and automatic generation of distributed code for embedded systems. We have presented a solution of software implemented fault-tolerance, which adapt the automatic code distribution algorithm of the "Algorithm Architecture Adequation" method (AAA). Basically, AAA takes as input a description of the algorithm to be distributed and a description of the target architecture. AAA first produces a static distributed schedule of a given algorithm onto a given distributed architecture, and then it generates a real-time distributed executive implementing this schedule.

Since we are dealing with embedded systems, we do not want to add redundant hardware. Rather, we choose to take advantage of the existing parallelism offered by the target distributed architecture. Also we consider only processor failures and assume they have a fail-silent behavior.

We are therefore given an algorithm specification, an architecture specification, some real-time constraints, and a number K of processor failures to be tolerated. Taking advantage of AAA, we have proposed a new scheduling heuristic that produces automatically a static distributed fault-tolerant schedule of the given algorithm onto the given distributed architecture.

Our solution is based on the software redundancy of the computation operations and on the time redundancy of the communications. When the main processor executing an operation fails, the results of one replica operation are sent, after some timeout, by a backup processor chosen at compile time. The implementation uses a scheduling heuristic for optimizing the critical path of the obtained distributed algorithm. The communication overhead is minimal; on the other hand, the occurrence of several failures in a row is not well supported. We have finally shown that if the given target architecture uses a single multi-point link, then intermittent fail-silent processor failures can also be treated.

The solution proposed here can fail, either if the real-time constraints can't be satisfied by the obtained distributed fault-tolerant schedule, or if less than K processor failures can be tolerated. This can happen if the intrinsic parallelism offered by the target architecture is not sufficient.

Finally, our solution can only tolerate processor failures. We are currently working on new solutions to tolerate also the communication link failures. We are also experimenting our method on an electric autonomous vehicle, with a 5 processors distributed architecture and a CAN bus.

Acknowledgments

The authors would like to thank Cătălin Dima, Thierry Grandpierre, Claudio Pinello, and David Powell for their helpful suggestions.

References

[1] T. Anderson and J.C. Knight. A framework for software fault-tolerance in real-time systems. *IEEE Trans. on Soft-*

ware Engineering, 9(3):355–364, May 1983.

[2] L. Arditi and A. Bouali. Using ESTEREL and formal methods to increase the confidence in the functional validation of a commercial DSP. In *ERCIM Workshop on Formal Methods for Industrial Critical Systems, FMICS'99*, Trento, Italy, 1999.

[3] J.-L. Bergerand and E. Pilaud. SAGA: A software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988.

[4] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone. ESTEREL: a formal method applied to avionic software development. *Science of Computer Programming*, 36:5–25, 2000.

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] 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.

[7] A.A. Bertossi, L.V. Mancini, and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 10:934–945, 1999.

[8] A. Bouali. XEVE: An ESTEREL verification environement. In *International Conference on Computer Aided Verification, CAV'98*, LNCS, Vancouver, Canada, June 1998. Springer-Verlag.

[9] M. Caccamo and B. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. In *5th International Conference on Real-Time Computing Systems and Applications*. IEEE Press, October 1998.

[10] J.-Y. Chung, J.W.S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. on Computers*, 39(9):1156–1174, September 1990.

[11] D. Powell et al. The Delta-4 approach to dependability in open distributed systems. In *18th IEEE International Symposium on Fault-Tolerant Computing, FTCS'88*, pages 246–251, Tokyo, Japan, June 1988. IEEE Computer Society Press.

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

[13] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerant scheduling on a hard real-time multiprocessor system. In H.J. Siegel, editor, *8th International Symposium on Parallel Processing*, pages 775–783, Los Alamitos, CA, April 1994. IEEE Computer Society Press.

[14] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. Research Report 4006, INRIA, September 2000.

[15] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

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

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

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

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

[20] S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'00*, volume 1926 of *LNCS*, pages 82–93, Pune, India, September 2000. Springer-Verlag.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[22] J.-C. Laprie et al. *Dependability: Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.

[23] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SYNDEX software environment for real-time distributed systems design and implementation. In *European Control Conference*, volume 2, pages 1684–1689. Hermès, July 1991.

[24] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[25] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. on Software Engineering*, 18(9):785–793, September 1992.

[26] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2):189–219, 1994. Research Report CSL-93-01.

[27] Y. Sorel. Massively parallel computing systems with real time constraints, the "algorithm architecture adequation" methodology. In *Massively Parallel Computing Systems Conference*, Ischia, Italy, May 1994.

[28] VERIMAG. *The Declarative Code DC*, 1.2b edition, June 1999. Available at http://www-verimag.imag.fr/ raoul/DC-WWW.

[29] A. Vicard. *Formalisation et Optimisation des Systèmes Informatiques Distribués Temps-Réel Embarqués*. PhD Thesis, University of Paris XIII, July 1999.