Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems *

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

Abstract

We present in this paper a heuristic for producing automatically a distributed fault-tolerant schedule of a given data-flow algorithm onto a given distributed architecture. The faults considered are processor failures, with a fail-silent behavior. Fault-tolerance is achieved with the software redundancy of computations and the time redundancy of data-dependencies.

Keywords: *Real-time embedded systems, software implemented fault-tolerance, static scheduling, distribution heuristics.*

1 Introduction

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, critical real-time, limited resources, and distributed and heterogeneous architectures.

Synchronous programming [4] offers specification methods and formal verification tools that give satisfying answers to the above mentioned needs. 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. However, two aspects extremely important w.r.t. the target fields, *distribution* and *fault-tolerance* [6], are not taken into account.

Starting from a data-flow algorithm and a distributed architecture, our goal is to produce automatically a fault-tolerant distributed schedule of the algorithm onto the architecture. The kind of failures that must be tolerated is fail-silent processor failures [1], and their number is also known. To achieve this, we present a distribution heuristics which replicates the computations and data-dependencies of the algorithm so that the obtained static schedule implements the software redundancy of computations and time redundancy of data-dependencies. By taking into account the execution durations of all computations on all processors, and the communication durations of all data-dependencies on all communication links, we are able to compute the total execution time of the obtained schedule, both in the presence and in the absence of faults.

Our solution extends the "Algorithm Architecture Adequation" method [3] with fault-tolerance, and allows us to use tools like SYNDEX [7] to produce automatically fault-tolerant distributed code for our target embedded systems.

2 Fault-Tolerance Problem and 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 time redundancy.

Algorithm Model. The algorithm is modeled by a *data-flow graph*. Each vertex is an *operation* and each edge is a *data-dependency*. 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 can be:

1. A computation operation (`comp`): output values depend only on input values, i.e., there is no internal state variable and no side effect.
2. A memory operation (`mem`): the data is held by a `mem` in sequential order between iterations.
3. An external input/output operation (`extio`): operations with no predecessor (resp. no successor) are external input interface (resp. output) handling the events produced by the sensors (resp. actuators). These are the only operations with side effects, but we assume that two executions of a given input `extio` in the same iteration always produce the same output value.

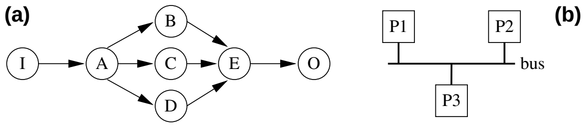

Figure 1. (a) An algorithm graph: I and O are `extios`, A–E are `comps`. (b) An architecture graph with three processors and a bus.

*This work has been funded by the TOLÈRE research action of INRIA. Extended abstract published in the *21st International Conference on Distributed Computing Systems*, Phoenix, USA, April 2001.

†INRIA-BIP. 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. Tel: +33 139 63 52 60. Email: Yves.Sorel@inria.fr

Architecture Model. The architecture is modeled by an hypergraph, where nodes are processors or communication links, and edges are connections between them. A processor is made of a CPU, a local memory, and several communication units, each connected to one communication link. Communication units execute data transfers, called `comms`.

Distribution Constraints. They consist in assigning to each pair $\langle$operation, processor$\rangle$ the value of the execution duration of this operation onto this processor. The value "∞" means that this operation cannot be executed on this processor. Similarly, we assign a communication duration to each pair $\langle$data dependency, communication link$\rangle$.

For instance, the distribution constraints for the algorithm graph and the architecture graph of Figure 1 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.

3 The Proposed Solution

Principle. We use the software redundancy of `comps`, `mems`, and `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 chosen as the *main replica*. Completion dates are computed according to the tables 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. 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, and another main processor executing a replica of o sends o's results to the successor operations. This raises the three following questions:

1. When is the main processor of an operation declared faulty? With a single multi-point link (e.g., a bus), the main processor of o broadcasts the o's outputs while the backup processors observe the bus 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 [5]. To deal with point-to-point links and to avoid heavy agreement algorithms, we have proposed in [2] another solution, based on the active redundancy on both `comps` and `comms`. In this solution there is no main replica to choose and no timeout to compute, but on the other hand, there is more communication overhead.

2. How are computed the timeouts associated to the communications? Each timeout is computed as the worst case upper-bound of the message transmission delay, w.r.t. the characteristics of the communication network. This is the least possible value avoiding multiple sendings of messages.

3. How is the new main processor selected after a failure? We choose the processor which finishes first the execution of the replica operation. For each operation, we 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.

We will evaluate the performances of our heuristic according to the following criteria:

1. Computation and communication overhead introduced by fault-tolerance.
2. Timing performances of the faulty system, i.e., a system presenting at least one failure.
3. Ability to support several failures in the same iteration.
4. Appropriateness to different kinds of architecture.

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 [3, 8].

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. Firstly, before the heuristic, we compute from the algorithm graph and the characteristic tables, 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. Secondly, at each step n of the heuristic, we compute 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.

At each step n, 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.

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:
$\sigma^{best}(n)(o) = \max_{o_i \in O_{impl}(n)} \cup_{l=1}^{l=K+1}\{\sigma^{opt}(n)(o_i, p_{i_l})\}$
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

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.

Fault-Tolerance Overhead. Each operation of the algorithm graph is replicated $K + 1$ times, but each replica only receives its inputs once, from the main replica of the predecessor operation. Thus, each data-dependency leads to at most $K + 1$ inter-processor communications. Indeed, when both 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.

When a failure occurs, we claim that the number of inter-processor communications in the resulting schedule is less than in the initial schedule. Let p be a faulty processor, and consider an operation o whose main replica is assigned to p. Among the message sent by o's main replica in the initial schedule, a number k_{intra} are intra-processor communications because the destination is also assigned to p. The number of inter-processor communications actually sent by o's main replica is k_{inter}. When p which fails, a new main processor is 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 Figure 1, with $K = 1$. The execution characteristics of each `comp`, `mem`, `extio`, and `comm` are specified by the tables of Section 2.

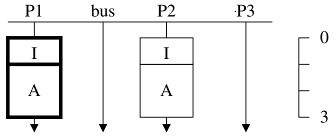

Figure 2. Temporary schedule: only I and A are scheduled.

After the first two steps, we get the temporary schedule of Figure 2. Each operation is a white box, whose height is proportional to its execution time, and main replicas are thicker. Each `comm` is a gray box, whose height is proportional to communication time, and whose ends are bound by arrows from the source to the destination operation.

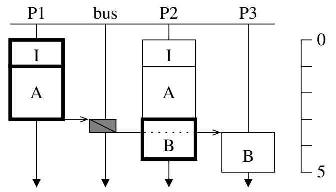

Figure 3. Temporary schedule: I, A, and B are scheduled.

At the next step, 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 4.5, so P2 is chosen as the main processor of B. Similarly, with P3 we get 5, so P3 is chosen as the backup processor. Finally, when all operations are scheduled, we get:

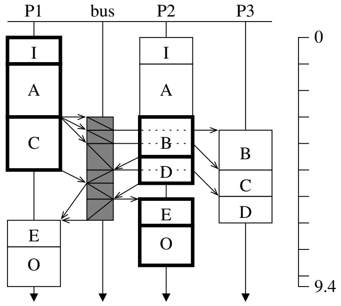

Figure 4. Final fault-tolerant schedule.

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.

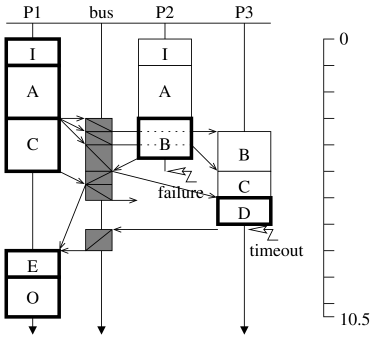

Figure 5. Timed execution when P2 crashes.

Analysis. The non fault-tolerant schedule produced for our example with the basic heuristic of SynDEx is:

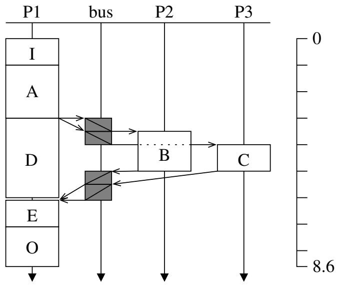

Figure 6. Non fault-tolerant schedule.

In this particular case, the overhead introduced by fault-tolerance 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). However, the replicas do not send their result until a failure occurs. Therefore the communication overhead is minimal.

When a failure occurs, extra communication can take place. This is the case of our example when P2 crashes (Figure 5). 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 [2] should be preferred. It also supports several failures during the same iteration.

4 Conclusion

We have presented a scheduling heuristic for obtaining a static distributed fault-tolerant schedule, starting from an algorithm specification, an architecture specification, some real-time constraints, and a number K of processor failures to be tolerated. It is based on the software redundancy of the computation operations and on the time redundancy of the communications. A replicated operation only sends its result, after some timeout, when the main processor running the same operation fails. 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.

Our solution 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 parallelism offered by the target architecture is not sufficient.

References

[1] 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.

[2] 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.

[3] 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.

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

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

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

[7] 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.

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