A Constraint Logic Programming Framework for the Synthesis of Fault-Tolerant Schedules for Distributed Embedded Systems

Kåre Harbo Poulsen, Paul Pop
Informatics and Mathematical Modelling Dept.
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark
s001873@student.dtu.dk, Paul.Pop@imm.dtu.dk

Viacheslav Izosimov
Computer and Information Science Dept.
Linköping University
SE-581 83 Linköping, Sweden
viaiz@ida.liu.se

Abstract

We present a constraint logic programming (CLP) approach for synthesis of fault-tolerant hard real-time applications on distributed heterogeneous architectures. We address time-triggered systems, where processes and messages are statically scheduled based on schedule tables. We use process re-execution for recovering from multiple transient faults. We propose three scheduling approaches, which each present a trade-off between schedule simplicity and performance, (i) full transparency, (ii) slack sharing and (iii) conditional, and provide various degrees of transparency. We have developed a CLP framework that produces the fault-tolerant schedules, guaranteeing schedulability in the presence of transient faults. We show how the framework can be used to tackle design optimization problems.The proposed approach has been evaluated using extensive experiments.

1. Introduction

Safety-critical applications must function correctly and meet their timing constraints even in the presence of faults. Such faults can be permanent (i.e. damaged hardware), transient (e.g. caused by electromagnetic interference), or intermittent (recurring). Transient faults are most common, and increasing so, due to the raising level of integration in semiconductors [5].

Researchers have proposed several hardware architecture solutions, such as MARS [16], TTA [15], and XBW [4], that use hardware redundancy to tolerate a single permanent fault. To use such schemes for tolerating transient faults, which are more numerous, incurs very large hardware cost. In this case, time-redundant approaches such as re-execution, replication and checkpointing are more appropriate.

The schedulability of an application can be guaranteed as well as appropriate levels of fault-tolerance achieved using preemptive online scheduling [2, 3, 10, 23]. However, such approaches lack the predictability required in safety-critical applications, where static off-line scheduling is the only option for ensuring both the predictability of worst-case behavior, and high resource utilization [15]. A simple heuristic for combining several static schedules to mask fault-patterns through replication is proposed in [6], however, without considering any timing constraints. This approach is used as basis for cost and fault-tolerance trade-offs within the Metropolis environment [20].

Fohler [8] proposes a method for joint handling of aperiodic and periodic processes by inserting slack for aperiodic processes in the static schedule, such that timing constraints of periodic processes are met. In [9] he extends this to cover fault-tolerance, considering overheads for several fault-tolerance techniques, including replication, re-execution and recovery blocks.

Kandasamy [14] proposes a list-scheduling technique for building a static schedule that can mask the occurrence of faults, making the re-execution transparent.

In [12] we have presented a list scheduling-based heuristic for the generation of fault-tolerant schedules. In [11] we have used a tabu-search meta-heuristic on top of list scheduling to optimize the assignment of fault-tolerance policies (i.e. re-execution vs. active replication) in order to reduce the fault-tolerance overheads. Such heuristics are able to produce good quality solutions in a reasonable time. Researchers have used *constraint logic programming* (*CLP*) [18, 17, 19, 7] in the context of system-level design, but not for fault-tolerance aspects. The advantages of *CLP* are: it can capture complex design constraints and trade-offs, it is flexible, general, and easy to extend.

In this paper we propose a *CLP* framework for producing fault-tolerant schedules such that the application is schedulable in the presence of transient faults, and the constraints and trade-offs imposed by the designer are satisfied. We show how the framework can be used to easily capture complex design optimization problems, e.g. fault-tolerance policy assignment.

The next section presents the system architecture and fault model. The application model is presented in section 3. Section 4 introduces the three scheduling approaches considered. We present the *CLP* implementation in section 5, and design optimization in 6. The experiments are presented in section 7.

2. Hardware Architecture and Fault-Model

An architecture consists of $\mathcal{N}$ heterogeneous processing elements connected by a bus. Processes, as well as messages, are statically scheduled. We consider that at most k transient faults may occur anywhere in the system during one operation cycle of the application. Thus, several faults may occur on the same, or different, processors k is determined as a design parameter, such that the desired reliability for the system is achieved.

We assume a combination of hardware and software-based error detection methods [13] to be part of the architecture. The software architecture, including real-time kernel, error detection and fault-tolerance mechanisms and communication bus are considered fault-tolerant. Worst-case execution times include fault-detection and recovery.

3. Application Model

An application $\mathcal{A}(\mathcal{V},\mathcal{E})$ is a set of directed, polar, acyclic graphs $\mathcal{G}_i(\mathcal{V}_i,\mathcal{E}_i) \in \mathcal{A}$, with a sink and a source node. Each node $P_i \in \mathcal{V}$ represents one process. Dependencies are denoted as $e_{ij} \in \mathcal{E}$, and a process is activated when all its inputs have arrived and issues its outputs when it terminates. Processes are non-preemptable. Communication between processes on the same process is part of the worst-case execution time, whereas communication between processes on different processors are passed over the bus. Each process P_i has a corresponding start-time S_{P_i}, and mapping M_{P_i}.

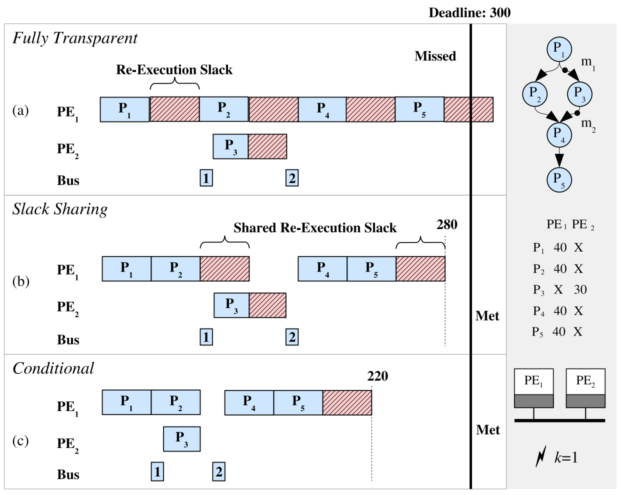


Figure 1: Scheduling strategies

4. Scheduling Strategies

Each node runs a real-time kernel which does process activation and message transmission based on the local schedule table. The initial schedule tables in the case of no-faults are called *root schedules*. When a fault is detected the kernel switches to an alternative schedule, the *contingency schedule*, which holds a schedule that will allow for re-executing the process and recovering. All contingency schedules are static. The worst-case delay of an application is given by the finishing time of its longest fault scenario.

Transparent recovery of processes, i.e. the operation of other processes are not affected, has the advantages of fault containment, good debugability and less memory needed to store the schedules, but, needs very large slacks to be scheduled, which may make the application unschedulable [12]. Part of scheduling is policy assignment, which is essentially mapping of re-executions, either on the same processor, *re-execution*, or a different processor, *passive replication*.

In the following we propose three scheduling schemes that each represent a different trade-off between transparency and performance. For an example system the no-fault scenario is shown in figure 1. An application of five processes is mapped on two processing elements. Processes P_1, P_2, P_4 and P_5 are mapped on PE_1, and P_3 is mapped on PE_2. Messages m_1 and m_2 are sent over the bus. The worst-case execution times on the corresponding processing elements are given in the table. In the examples we consider $k = 1$.

4.1. Fully Transparent Scheduling

The simplest approach to build a fault tolerant schedule is to use transparent recovery. In this scheme a recovery slack of length kC_{P_i} is inserted after each process. This gives the online scheduler time to re-execute a failing process upto k times, without violating the timing of other processes.

However, the fully transparent approach incurs long delays which can make the application unschedulable, as is the case in figure 1(a).

4.2. Slack Sharing Scheduling

To reduce the end-to-end delay, we allow processes on the same processor to share the recovery slacks. In the schedule we see that, e.g. processes P_4 and P_5 share recovery slack. The size of the recovery slack is long enough to accommodate the k re-executions of any the processes that share it.

The slack sharing approach sacrifices some of the transparency in order to reduce the delays. In this scheduler, fault information is shared on the local processor, but faults are still transparent between processors. In this way, process P_3 has to wait until time 90 to start, to ensure that, if a fault has happened in P_1, process P_1 has had time to recover.

The slack sharing scheduling approach has all of the advantages of the fully transparent scheduler, but is able to reduce the amount of slack to be scheduled.

4.3. Conditional Scheduling

To get better performance it is necessary to trade-off all transparency, i.e. even processes on other processors may be affected by faults on a processor, and only exactly k recoveries are scheduled for any fault scenario. This scheduling scheme is similar to online scheduling, but, since all contingency schedules are calculated in advance, predictability and schedulability are achieved.

In this case, the online schedulers in the nodes will have to share information on fault occurrence. This allows the schedulers to respond efficiently to faults, and hence the delay is further reduced. Thus, the schedulers will rely on a *conditional schedule table*, which contains start times for each fault scenario.

This scheduling approach reduces the overheads due to fault-tolerance, by adapting online to the actual fault scenario. However, the time to broadcast the fault occurrence information on the bus is not negligible, and the time needed to derive the conditional schedule table offline, and the size for the stored schedules grow exponentially with k.

5. CLP-Based Scheduling

For *NP*-hard problems *CLP* has very good performance, and is hence an ideal platform for scheduling. In *CLP*, systems are described by a set of logic constraints which define valid conditions for the system variables. A solution to the modelled problem is an enumeration of all system variables, such that there are no conflicting constraints. We have implemented our synthesis approach using the ECL^{i}PS^e *CLP* system [1].

The logic constraints used in our model fall in four categories: (i) precedence, (ii) resources, (iii) timing, and (iv) fault tolerance. The three first have been previously addressed by researchers [18, 7], and we shall focus on the constraints for fault-tolerance. The constraints for the first two schemes are applied on the process graphs $\mathcal{G}_i \in \mathcal{A}$. The last scheme uses fault-tolerant process graphs (*FT-PG*) [11, 12]. These are graphs that capture all possible fault scenarios, using guarded transitions.

5.1. Fully Transparent Scheduling

In the fully transparent scheme, recovery slack is scheduled after each process. Hence no process may run until after $k+1$ executions of its precedents:

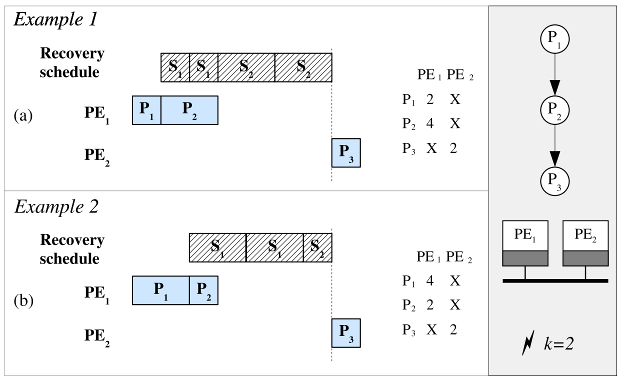
$$S_{P_j} \geq \forall_{e_{ij}} S_{P_i} + C_{P_i}(1+k), \text{ for all } P_j \in \mathcal{A} \quad (1)$$

5.2. Slack-Sharing Scheduling

For the slack sharing scheduler, processes with dependencies on the same and different processors need to be treated different.

Processes on the Same Processing Element. Processes executed on the same processor share recovery slack. This slack is scheduled after the root processes, thus is expressed simply as:

$$M_{P_i} = M_{P_j} \quad (2)$$



	PE1	PE2
P1	2	X
P2	4	X
P3	X	2

	PE1	PE2
P1	4	X
P2	2	X
P3	X	2

Figure 2: Special cases for slack sharing constraints.

Processes on Different Processing Elements. For processes on different processors a receiving process cannot be started until the recovery of its predecessors on all other processors is guaranteed. The situation where two processes on different processors have to communicate, can be split into two special cases.

Example 1: Consider the dependency between processes P_2 and P_3. Figure 2(a) shows the critical recovery path, when $C_{P_2} > C_{P_1}$. This determines when P_3 can safely be started:

$$S_{P_3} \geq S_{P_2} + C_{P_2}(1+k) \tag{3}$$

Example 2: In figure 2(b), where $C_{P_2} < C_{P_1}$m the availability of data is not only determined by the sending process, but also the process scheduled before:

$$S_{P_3} \geq S_{P_1} + C_{P_1}(1+k) + C_{P_2} \tag{4}$$

To generalize the shown constraints, in a way that can be used in a *CLP* model, detailed information of the longest recovery path is needed. This is achieved by creating a separate recovery schedule for the recovery processes.

The recovery schedule is set up in the following way. For each process P_i, a recovery process S_i is inserted into the recovery schedule with an edge e_{P_i,S_i}. In the recovery schedule, the same precedence and resource constraints as for the normal schedule are imposed. The finishing times of the processes in the recovery schedule are described by (F is the finishing time of a process):

$$F_{S_i} \geq S_{P_i} + C_{P_i}(1+k)$$
$$\wedge\, F_{S_i} \geq S_{S_i} + C_{P_i} \tag{5}$$

Using the recovery schedule, the general logic constraint for processes on different processors can now be written:

$$S_{P_j} \geq F_{S_i} \tag{6}$$

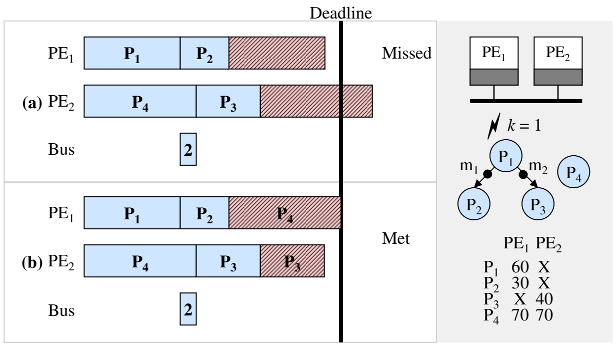
which leads to a general constraint for slack sharing:

$$M_{P_i} = M_{P_j} \wedge S_{P_j} \geq S_{P_i} + C_{P_i}$$
$$\vee\, S_{P_j} \geq F_{S_i}, \text{ for all } P_i \in \mathcal{A} \tag{7}$$

5.3. Conditional Scheduling

The conditional edges in the *FT-PG* form mutually exclusive fault-scenarios. As a consequence, two processes, which depend on mutually exclusive conditions (determined by the function *MutuallyExclusive*), will never be active in the same scenario. Hence, processes which are part of different scenarios can be scheduled on the same resource at the same time. This addition to the *resource constraint* is written as:

$$MutuallyExclusive(P_{i,j}, P_{l,n})$$
$$\vee\, M_{P_{i,j}} \neq M_{P_{l,n}}$$
$$\vee\, S_{P_{i,j}} \geq S_{P_{l,n}} + C_{P_{l,n}}$$
$$\vee\, S_{P_{l,n}} \geq S_{P_{i,j}} + C_{P_{i,j}}, \text{ for all } P_j \in \mathcal{A} \tag{8}$$



	PE1	PE2
P1	60	X
P2	30	X
P3	X	40
P4	70	70

Figure 3: Fault-tolerance policy optimization

6. Design Optimization Problems

The framework can capture complex design constraints, and handle several interesting design optimization problems.

Consider the example in figure 3, where the mapping of P_1, P_2 is fixed on PE_1, and P_3 is mapped on PE_2. These processes all use re-execution to tolerate faults, and we wish to decide mapping and fault-tolerance policy for P_4. The best mapping for P_4 is on PE_2, because then it can run in parallel to P_1. If we use re-execution for P_4, the shared re-execution slack for P_4 and P_3 will miss the deadline, as we can see in figure 3(a). However, if we re-execute P_4 on PE_1 instead (we say that P_4 is passively replicated on PE_1), as depicted in 3(b), the deadline is net. Using passive replication, the inputs of a process P_i have to be broadcast on the bus, and recovery slack on PE_1 is enlarged from 60 ms to 70 ms to accommodate P_4. Even with these overheads associated to passive replication, the overall delay is reduced and the deadline is met.

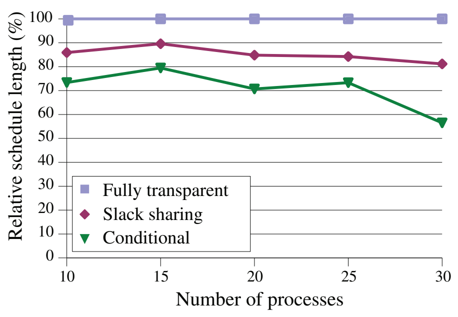
7. Experimental Results

Using *TGFF* (*task graphs for free*) 10 synthetic applications for each graph size 10, 15, 20, 25, and 30 processes are generated. A random 50% of the processes are made fault-tolerant, the other half is considered non-critical. A random mapping on an architecture consisting of three processing elements is applied. To focus on fault-tolerance aspects we disregard messages. Experiments are run on a ECL^{i}PS^e *CLP* system, version 5.10_44 on 2.2 GHz AMD 64-bit machines, with progressive time-out set for each run, based on the application size, to 15, 30, 45, 60 and 75 minutes, respectively.

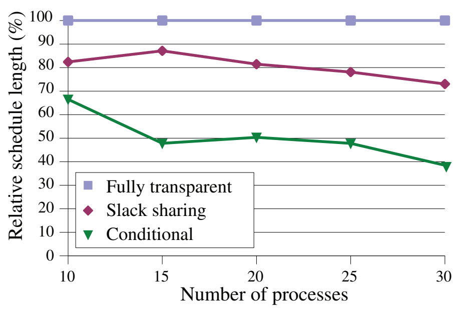
Firstly, we compare the three scheduling approaches in terms of the length of the schedules produced. The results are shown in figure 4(a)-(b) for $k = 1$ and $k = 2$. The x-axis marks the size of the graphs, and the y-axis is the schedule length, relative to the schedule produced using the fully transparent approach. The plotted points are the average of the ten graphs generated for each graph size.

From the graphs we see that for $k = 1$ the slack sharing scheme produces results that are 10-15% faster, and the conditional scheme does 20-30% better. This tendency is even more obvious for $k = 2$, where the slack sharing scheduling performs 20% better, and the conditional scheduling is 50% better than the transparent scheduling. The implementation is not able to find any solutions for graph sizes larger than 20 when doing conditional scheduling. This is due to the size of the internally used *FT-PG*s which grows exponentially with k [21].

Secondly, we wish to compare the length of the produced schedules with those produced by the *FT-PG list scheduling* heuristic we have presented in [12]. The applications are ran-



(a) Scheduling performance for $k = 1$



(b) Scheduling performance for $k = 2$

	Application size		
	10	15	20
$k = 1$	30.95%	27.31%	40.30%
$k = 2$	44.15%	50.40%	64.83%

(d) Comparison of *CLP* conditional with list scheduling

	Scheduling strategy		
k	**Fully Trans.**	**Slack Sharing**	**Conditional**
$k = 1$	1103796	919280	818585
$k = 2$	1655694	1286662	1085272

(c) Scheduling performance for the *mp3*-decoder.

Figure 4: Experimental Results

domly mapped on an architecture with 4 processors, and all processes are made redundant. The results are shown in the table in figure 4(d) for 10, 15 and 20 processes. We see that list scheduling results are far from optimal. In fact, for $k = 2$ the optimal schedules are as much as 65% faster.

We have evaluated our *CLP* approach on a real-life application, the *mp3*-decoder presented in [22]. The results are shown in the table in figure 4(c), where finishing times are expressed as number of cycles. Using slack sharing scheduling gives a performance increase of 17% and 23% for $k = 1$ and 2, respectively, compared to the fully transparent approach. For conditional scheduling, these numbers are 26% and 35%.

8. Conclusions

In this paper we have addressed fault-tolerant applications mapped on time-triggered embedded systems where both processes and messages are statically scheduled. We have proposed three scheduling approaches to the synthesis of fault-tolerant schedules, which provide trade-offs in terms of performance, memory consumption, runtime overhead and debugability.

The proposed strategies have been implemented using a *CLP* framework. The framework produces the fault-tolerant schedules such that the application is schedulable in the presence of transient faults, and is able to capture complex design constraints and design optimization problems such as mapping and fault-tolerance policy assignment.

The evaluations show that the *CLP* framework can find optimal solutions even for large applications, and outperforms the previously proposed list scheduling-based heuristics.

References

[1] K. Apt and M. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[2] A. Bertossi and L. Manchini. Scheduling algorithms for fault-tolerance in hard-real time systems. *Real Time Systems*, 7(3):229–256, 1994.

[3] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proc. of the Euromicro Workshop on Real-Time Systems*, pages 29–33, 1996.

[4] V. Claeson, S. Poldena, and J. Söderberg. The xbw model for dependendable real-time systems. In *Proc. of the Paralell and Distributed Systems Conf.*, pages 130–138, 1998.

[5] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23:14–19, 2003.

[6] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *Proc. of the Euromicro Parallel and Distributed Processing Workshop*, pages 410–417, 2001.

[7] C. Ekelin. *An Optimization Framework for Scheduling of Embedded Real-Time Systems*. PhD thesis, Chalmers University of Technology, 2004.

[8] G. Fohler. Joint scheduling of distributed complex periodic and hard aperidic tasks in statically scheduled systems. In *Proc. of the Real-Time Systems Symposium*, pages 152–161, 1995.

[9] G. Fohler. Adaptive fault-tolerance with statically scheduled real-time systems. *Proc. of the Euromicro Real-Time Systems Workshop*, pages 161–167, 1997.

[10] C. C. Han, K. G. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEE Trans. on Computers*, 52(3):362–372, 2003.

[11] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time- and cost-constrainted fault-tolerant distributed embedded systems. In *Proc. of the Design, Automation and Test in Europe Conf.*, pages 864–869, 2005.

[12] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant schedules with transparency/perofrmance trade-offs for distributed embedded systems. In *Proc. of the Design, Automation and Test in Europe Conf.*, pages 706–711, 2006.

[13] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley, 1989.

[14] N. Kandasamy, J. P. Hayes, and B. T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Trans. on Computers*, 52(2), February 2003.

[15] H. Kopetz. *Real-Time Systems-Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[16] H. Kopetz, A. Damm, C. Koza, and other. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9:25–40, 1989.

[17] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *In ACM Trans. on Design Automation of Electronic Systems*, 8(3):355–383, 2003.

[18] K. Kuchinski. Constraint-driven design space exploration for distributed embedded systems. *Journal of Systems Architecture*, 47:241–261, 2001.

[19] J. Liu, P. H. Chou, N. Bagerzadeh, and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *Proc. of the 9th Int. Symposium on Hardware/Software Codesign*, pages 153–158, 2001.

[20] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Proc. of the Design, Automation and Test in Europe Conf.*, pages 1164–1169, 2004.

[21] K. H. Poulsen. Reliability-aware energy optimisation for fault-tolerant embedded mp-socs. Master's thesis, DTU, 2007.

[22] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publishers, 2004.

[23] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of the Design, Automation and Test in Europe Conf.*, pages 918–923, 2003.