

# Scheduling and Voltage Scaling for Energy/Reliability Trade-offs in Fault-Tolerant Time-Triggered Embedded Systems

Paul Pop<sup>1</sup>, Kåre Harbo Poulsen<sup>1</sup>, Viacheslav Izosimov<sup>2</sup>, Petru Eles<sup>2</sup>

<sup>1</sup>Paul.Pop@imm.dtu.dk, s001873@student.dtu.dk  
Informatics and Mathematical Modelling Dept.  
Technical University of Denmark  
DK-2800 Kongens Lyngby, Denmark

<sup>2</sup>{viaiz | petel}@ida.liu.se  
Computer and Information Science Dept.  
Linköping University  
SE-581 83 Linköping, Sweden

## ABSTRACT

*In this paper we present an approach to the scheduling and voltage scaling of low-power fault-tolerant hard real-time applications mapped on distributed heterogeneous embedded systems. Processes and messages are statically scheduled, and we use process re-execution for recovering from multiple transient faults. Addressing simultaneously energy and reliability is especially challenging because lowering the voltage to reduce the energy consumption has been shown to increase the transient fault rates. In addition, time-redundancy based fault-tolerance techniques such as re-execution and dynamic voltage scaling-based low-power techniques are competing for the slack in the schedules. Our approach decides the voltage levels and start times of processes and the transmission times of messages, such that the transient faults are tolerated, the timing constraints of the application are satisfied and the energy is minimized. We present a constraint logic programming-based approach which is able to find reliable and schedulable implementations within limited energy and hardware resources.*

## Categories and Subject Descriptors

B.8.2 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance; C.3 [Special-Purpose and Application-Based Systems]: Real-Time and Embedded Systems

## General Terms

Algorithms, Design, Performance, Reliability

## Keywords

Embedded systems, Energy minimization, Reliability, Scheduling

## 1. INTRODUCTION

Safety-critical applications have to function correctly, meet their timing constraints and be energy-efficient even in the presence of faults. Such faults can be permanent (i.e., damaged microcontrollers or communication links), *transient* (e.g., caused by electromagnetic interference), or intermittent (appear and disappear repeatedly). The transient faults are the most common, and their number is increasing due to the raising level of integration in semiconductors.

Researchers have proposed several hardware architecture solutions, such as MARS [16], TTA [15] and XBW [6], that rely on hardware replication to tolerate a single permanent fault in any of the components of a fault-tolerant unit. Such approaches can be used for tolerating transient faults as well, but they incur very large hardware cost if the number of transient faults is larger than one. An alternative to such purely hardware-based solutions are approaches such as re-execution, replication, checkpointing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.  
Copyright 2007 ACM 978-1-59593-824-4/07/0009...\$5.00.

Several researchers have shown how the schedulability of an application can be guaranteed at the same time with appropriate levels of fault-tolerance using pre-emptive online scheduling [2, 3]. Considering the high degree of predictability of time-triggered systems [15], researchers have also proposed approaches for integrating fault-tolerance into the framework of static scheduling. A simple heuristic for combining several static schedules in order to mask fault-patterns through replication is proposed in [7], without, however, considering any timing constraints. This approach is used as the basis for cost and fault-tolerance trade-offs within the Metropolis environment [19].

Fohler [9] proposes a method for joint handling of aperiodic and periodic processes by inserting slack for aperiodic processes in the static schedule, such that the timing constraints of the periodic processes are guaranteed. In [10] he equates the aperiodic processes with fault-tolerance techniques that have to be invoked on-line in the schedule table slack to handle faults. Overheads due to several fault-tolerance techniques, including replication, re-execution and recovery blocks, are evaluated. When re-execution is used in a distributed system, Kandamany [14] proposes a list-scheduling technique for building a static schedule that can mask the occurrence of faults, making the re-execution transparent. Slacks are inserted into the schedule in order to allow the re-execution of processes in case of faults. The faulty process is re-executed, and the processor switches to an alternative schedule that delays the processes on the corresponding processor, making use of the slack introduced. The authors propose an algorithm for reducing the necessary slack for re-execution.

Applying such fault-tolerance techniques introduces overheads in the schedule and thus can lead to unschedulable systems. Very few researchers [14, 19] consider the optimization of implementations to reduce the overheads due to fault-tolerance and, even if optimization is considered, it is very limited and considered in isolation, and thus is not reflected at all levels of the design process, including mapping, scheduling and energy minimization [11].

Regarding energy minimization, the most common approach that allows energy/performance trade-offs during run-time of the application is dynamic voltage scaling (DVS) [20]. DVS aims to reduce the dynamic power consumption by scaling down operational frequency and circuit supply voltage. There has been a considerable amount of work on dynamic voltage scaling. For a survey of the topic, the reader is directed to [20].

Incipient research has analyzed the interplay of energy/performance trade-offs and fault-tolerance techniques [4, 8, 18, 22]. Time-redundancy based fault-tolerance techniques (such as re-execution and checkpointing) and dynamic voltage scaling-based low-power techniques are competing for the slack, i.e., the time when the processor is idle. The interplay of power management and fault recovery has been addressed in [18], where checkpointing policies are evaluated with respect to energy. In [8] time redundancy is used in conjunction with information redundancy, which does not compete for

slack with DVS, to tolerate transient faults. In [22] fault tolerance and dynamic power management is studied, and rollback recovery with checkpointing is used in order to tolerate multiple transient faults in the context of distributed systems.

Addressing simultaneously energy and reliability is especially challenging because lowering the voltage to reduce energy consumption has been shown to exponentially increase the number of transient faults [23]. The main reason for such an increase is that, for lower voltages, even very low energy particles are likely to create a critical charge that leads to a transient fault. However, this aspect has received very limited attention. Zhu [24] has proposed a reliability-aware DVS greedy heuristic for single processors, while in [23] a single-task checkpointing scheme is evaluated.

In [11] we have shown how re-execution and active replication can be combined in an optimized implementation that leads to a schedulable fault-tolerant application without increasing the amount of employed resources. In [12] we have addressed transparency/performance trade-offs during the synthesis of fault-tolerant schedules. In this paper, we consider a very different trade-off, namely, energy versus reliability. To the best of our knowledge, we are the first to address such a trade-off in the context of multiprocessor embedded systems.

We consider heterogeneous distributed time-triggered systems, where both processes and messages are statically scheduled. The transient faults are tolerated through process re-execution by switching to pre-determined contingency schedules. In this context, we propose an approach to the scheduling and voltage scaling that decides the voltage levels and start times of processes and the transmission times of messages, such that the transient faults are tolerated, the timing constraints of the application are satisfied and the energy consumption in the no-fault scenario is minimized. We propose a novel constraint logic programming-based algorithm for the synthesis of fault tolerant schedules that takes into account the influence of voltage scaling on reliability.

The next two sections present the system architecture and the application model, respectively. The energy and reliability models are presented in Section 4. Section 5 presents our problem formulation, the scheduling strategies considered, and a motivational example. Section 6 outlines the proposed scheduling and voltage scaling approach. The evaluation of the proposed strategies is presented in Section 7.

## 2. SYSTEM MODEL

We consider architectures composed of a set  $\mathcal{N}$  of DVS-capable nodes which share a broadcast communication channel. The communication channel is statically scheduled such that one node at a time has access to the bus, according to the schedule determined off-line.

We have designed a software architecture which runs on the CPU in each node, and which has a real-time kernel as its main component. The processes activation and message transmission is done based on the local schedule tables.

In this paper we are interested in fault-tolerance techniques for tolerating transient faults, which are the most common faults in today's embedded systems. We have generalized the fault-model from [14] that assumes that one single transient fault may occur on any of the nodes in the system during the application execution. In our model, we consider that at most  $k$  transient faults may occur anywhere in the system during one operation cycle of the application. Thus, not only several transient faults may occur simultaneously on several processors, but also several faults may occur on the same processor. The number of  $k$  transient faults correspond to a reliability goal  $R_g$ . We consider that if the system reliability  $R_S$  drops below the required reliability  $R_g$ , then the assumption of only  $k$  transient faults occurring is no longer valid, i.e.,  $k' > k$  faults are likely to occur instead.

The error detection and fault-tolerance mechanisms are part of the software architecture. We assume a combination of hardware-based (e.g., watchdogs, signature checking) and software-based error detection methods, systematically applicable without any knowledge of the application (i.e., no reasonableness and range checks) [13]. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant. In addition, we assume that message fault-tolerance is achieved at the communication level, for example through hardware replication of the bus.

We use re-execution for tolerating faults. Let us consider a process  $P_1$  and a fault-scenario consisting of  $k = 2$  transient faults that can happen during one cycle of operation. The first execution runs at a voltage level  $V$ . In the worst-case fault scenario, the first fault happens during the process  $P_1$ 's first execution, and is detected by the error detection mechanism, after a worst-case *error detection overhead*  $\alpha_1$ . Once the error has been detected, the task has to be recovered. After a worst-case *recovery overhead* of  $\mu_1$ ,  $P_1$  will be executed again. Since we concentrate on minimizing the energy consumption in the no-fault scenario, we consider that all the re-executions are performed at full speed, i.e., at the maximum voltage  $V_{max}$  and frequency  $f_{max}$ . Its second execution in the worst-case could also experience a fault. Finally, the third execution of  $P_1$  will take place without fault. Note that the overheads  $\alpha_i$  and  $\mu_i$  for a process  $P_i$  are considered as part of its worst-case execution time  $C_i$ .

## 3. APPLICATION MODEL

We model an application  $\mathcal{A}(\mathcal{V}, \mathcal{E})$  as a set of directed, acyclic, polar graphs  $\mathcal{G}_k(\mathcal{V}_k, \mathcal{E}_k) \in \mathcal{A}$ . Each node  $P_i \in \mathcal{V}$  represents one process. An edge  $e_{ij} \in \mathcal{E}$  from  $P_i$  to  $P_j$  indicates that the output of  $P_i$  is the input of  $P_j$ . A process can be activated after all its inputs have arrived and it issues its outputs when it terminates. Processes are non-preemptable and thus cannot be interrupted during their execution. Fig. 1 depicts an application  $\mathcal{A}$  consisting of a graph  $\mathcal{G}_1$  with five processes,  $P_1$  to  $P_5$ .

The communication time between processes mapped on the same processor is considered to be part of the process worst-case execution time and is not modeled explicitly. Communication between processes mapped to different processors is performed by message passing over the bus. Such message passing is modeled as a communication process inserted on the arc connecting the sender and the receiver process, and is depicted with black dots in the graph in Fig. 1.

The mapping of a process in the application is determined by a function  $\mathcal{M}: \mathcal{V} \rightarrow \mathcal{N}$  where  $\mathcal{N}$  is the set of nodes in the architecture. For a process  $P_i \in \mathcal{V}$ ,  $\mathcal{M}(P_i)$  is the node to which  $P_i$  is assigned for execution. We consider that the mapping is given, and we know the worst-case execution time  $C_i$  of process  $P_i$ , when executed on  $\mathcal{M}(P_i)$ . In Fig. 1, the mapping is given in the table besides the application graph. We also consider that the size of the messages is given.

All processes and messages belonging to a process graph  $\mathcal{G}_i$  have the same period  $T_i = T_{\mathcal{G}_i}$  which is the period of the process graph. If communicating processes are of different periods, they are combined into a merged graph capturing all activations for the hyper-period (LCM of all periods). A deadline  $D_{\mathcal{G}_i} \leq T_{\mathcal{G}_i}$  is imposed on each process graph  $\mathcal{G}_i$ .

## 4. ENERGY AND RELIABILITY MODELS

We use the power model from [5] which shows that by varying the circuit supply voltage  $V_{dd}$ , it is possible to trade-off between power consumption and performance. The reliability  $R_i$  of a process  $P_i$  is defined as the probability of its successful execution, and it is captured by the exponential failure law [13]:

$$R_i = e^{-\lambda C_i} \quad (1)$$

where the term  $\lambda$  is the failure rate, which describes the amount of errors that will occur per second. For a system with the ability to handle

$k$  faults, a process will have to be able to perform  $k$  redundant recovery executions. For such a setup, the reliability is given by the probability of *not* all recoveries failing. Formally, this is expressed as [13]:

$$R_i' = 1 - (1 - R_i)^{1+k} \quad (2)$$

where the last term is the probability of all processes failing in the same run. For independent faults, the reliability of  $\mathcal{A}$  is [13]:

$$R_{\mathcal{A}} = \prod_{P_i \in \mathcal{A}} R_i' \quad (3)$$

#### 4.1 Energy/Reliability Trade-off Model

The equations presented so far do not account for the influence of voltage on reliability. However, lowering the voltage has been shown to dramatically lower the reliability [23]. Thus, the failure rate  $\lambda$  of a system is dependent on the voltage the system is run at. The relation between the two can be described by the expression proposed in [23]:

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \quad (4)$$

in which  $\lambda_0$  is the failure rate of the processor when run at maximum voltage  $V_{max}$  and frequency  $f_{max}$ , and  $d$  is an architecture specific constant. The variable  $f$  denotes the scaling factor, capturing both the voltage and the corresponding change in frequency. Equation 4 considers normalized voltages, i.e.,  $V_{max}$  is assumed to be 1. Thus, for a scaling factor  $f$ , the corresponding supply voltage is  $V_{dd} = f \times V_{max} = f$ . Regarding frequency, for a small threshold voltage  $V_r$ , the circuit delay becomes  $1/V_{dd} = 1/f$ .

Let us now return to the application reliability derived previously. Thus, the reliability of a single process, scaled with a factor  $f$  is:

$$R_i^f = e^{-\lambda(f)C_i/f} \quad (5)$$

We can now update Equation 2 that captures the reliability of a process  $P_i$ . Considering that all  $k$  recoveries are running at full speed (at  $f_{max}$ , with a corresponding reliability  $R_i^0$ ), the reliability of  $P_i$  is:

$$R_i^f(f) = 1 - (1 - R_i^0)^k (1 - R_i^f) = 1 - (1 - e^{-\lambda_0 C_i})^k (1 - e^{-\lambda(f) C_i / f}) \quad (6)$$

## 5. PROBLEM FORMULATION

The problem we are addressing in this paper can be formulated as follows. Given an application  $\mathcal{A}$  with a reliability goal  $R_g$  corresponding to  $k$  transient faults, mapped on an architecture consisting of a set of hardware nodes  $\mathcal{N}$  (interconnected via a broadcast bus  $B$ ), we are interested to determine the schedule tables  $\mathcal{S}$  such that the application is fault-tolerant, schedulable, and the energy of the no-fault execution scenario is minimized. Note that deciding on the schedule tables  $\mathcal{S}$  implies deciding on both the start times and the voltage levels for each process.

### 5.1 Scheduling Strategies

The scaled execution of processes and the recovery slack needed for re-executions introduce delays that can violate the timing constraints of the application. In addition, reducing the voltage to decrease the energy consumption, has a negative impact on the application reliability.

Let us consider the example in Fig. 1, where we have an application consisting of five processes,  $P_1$  to  $P_5$  and two messages,  $m_1$  and  $m_2$ , mapped on an architecture with two processors,  $N_1$  and  $N_2$ . Processes  $P_1, P_2, P_4$  and  $P_5$  are mapped on  $N_1$ , and  $P_3$  is mapped on  $N_2$ . Message  $m_1$  is sent from  $P_1$  to processes  $P_3$  and  $m_2$  from  $P_3$  to  $P_4$ . The worst-case execution times of each process on its corresponding processor are depicted in the table, and the deadline of the application, depicted with a thick vertical line, is 215 ms. The voltage levels for each processor and reliability numbers ( $d$  and  $\lambda_0$ ) are also given in the figure. We consider a reliability goal  $R_g = 0.999\,999\,990$  and  $k = 1$ .

In Fig. 1 we illustrate three scheduling strategies: (a) full transparency, (b) transparent recovery and (c) no transparency. For each alternative, on the left side (a<sub>1</sub>–c<sub>1</sub>) we show the shortest possible schedule when no faults occur and no voltage scaling is performed. In case no voltage scaling is performed, the energy consumption  $E_{\mathcal{A}}^0$  and reliability  $R_{\mathcal{A}}^0$  of application  $\mathcal{A}$  do not depend on the scheduling strategy. We would like to determine in a<sub>2</sub>–c<sub>2</sub> schedules that minimize the energy consumption, and meet the deadline, even in the presence of faults. The right side (a<sub>3</sub>–c<sub>3</sub>) depicts the worst-case fault scenario (resulting in the longest schedule) corresponding to the minimal energy schedules in a<sub>2</sub>–c<sub>2</sub>.

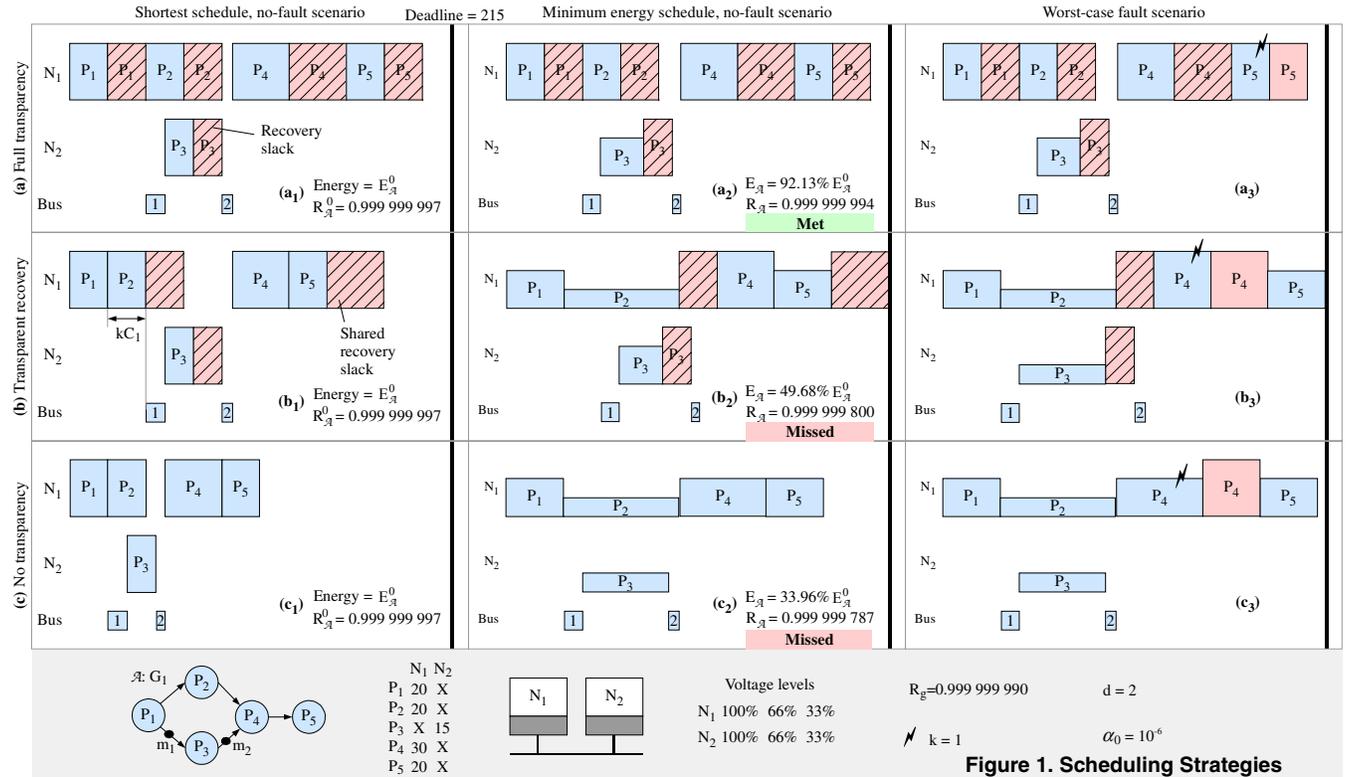


Figure 1. Scheduling Strategies

Whenever a fault occurs, the faulty process has to be re-executed. Thus, the scheduler in a processor that experiences a fault has to switch to another schedule, containing a different start time for that process. For example, according to the schedule in Fig. 1a<sub>1</sub>, processes are scheduled at the times indicated by the light blue rectangles in the Gantt chart. Once a fault occurs in  $P_5$ , the scheduler on node  $N_1$  will have to switch to another schedule, illustrated in Fig. 1a<sub>3</sub>, where  $P_5$  is delayed with  $C_5$  to account for the fault.

All the alternative schedules needed to run the application in case of faults are produced off-line by a scheduling algorithm. The end-to-end worst-case delay of an application is given by the maximum finishing time of any schedule, since this is a situation that can happen in the worst-case scenario. For the application in Fig. 1a<sub>1</sub>, the largest delay is produced by the schedule depicted in Fig. 1a<sub>3</sub>, which has to be activated when a fault happens in  $P_5$ .

Depending on how the schedule table is constructed, the re-execution of a process has a certain impact the execution of other processes. In Fig. 1a<sub>1</sub>, we have constructed the schedule such that each execution of a process  $P_i$  is followed by a *recovery slack*, which is idle time on the processor, needed to recover (re-execute) the failed process. For example, for  $P_3$  on node  $N_2$ , we introduce a recovery slack of  $k \times C_3 = 15$  ms to make sure we can recover  $P_3$  even in the case it experiences the maximum number of  $k$  faults. Thus, a fault occurrence that leads to the re-execution of any process  $P_i$  will only impact  $P_i$ , and not the other processes, since the re-execution time for  $P_i$  is already accounted in the schedule table. We call such an approach *fully transparent*, because fault occurrence in a process is transparent to *all* other processes, on the same, or on other processors.

The minimum energy in this case, corresponding to the schedule in Fig. 1a<sub>2</sub> is 92.13% of  $E_{\mathcal{A}}^0$ . Note that this minimum energy schedule has to meet the deadline even in the worst-case fault scenario. Due to this, we are able to scale only  $P_3$ , at 66%, without missing the deadline. For such a limited voltage reduction, the reliability, calculated according to Equations (3) and (6), meets the reliability goal. Although transparency has the advantages of fault containment, improved debugability and less memory needed to store the fault-tolerant schedules [12], it will, however, introduce large delays that can violate the timing constraints of the application, and reduce the slack available for voltage scaling.

We would like a scheduling approach where the delay is reduced, thus increasing the slack available for voltage scaling. The straightforward way to reduce the end-to-end delay is to share the recovery slacks among several processes. For example, in Fig. 1b<sub>1</sub>, processes  $P_4$  and  $P_5$  share the same recovery slack on processor  $N_1$ . This shared slack has to be large enough to accommodate the recovery of the largest process (in our case  $P_4$ ) in the case of  $k$  faults. This slack can then handle  $k$  faults also in  $P_5$ , which takes less to execute than  $P_4$ .

In Fig. 1b we consider an approach called *transparent recovery*, where the fault occurring on one processor is masked to the other processors in the system, but not, as in the case with full transparency, within a processor. Thus, on a processor  $N_i$  where a fault occurs, the scheduler has to switch to an alternative schedule that delays the descendants of the faulty process running on  $N_i$ . However, a fault happening on another processor, is not visible on  $N_i$ , even if the descendants of the faulty process are mapped on  $N_i$ . For example, in Fig. 1b<sub>1</sub>, where we assume that no faults occur, in order to isolate node  $N_2$  from the occurrence of a fault on node  $N_1$ , message  $m_1$  from  $P_1$  to  $P_3$ , cannot be transmitted at the end of  $P_1$ 's execution.  $m_1$  has to arrive at the destination at a fixed time, regardless of what happens on node  $N_1$ , i.e., transparently. Consequently, the message can only be transmitted after a time  $k \times C_1$ , at the end of the recovery of  $P_1$  in the worst-case scenario. However, a fault in  $P_1$  will delay process  $P_2$  which is on the same processor.

This approach will lead to a reduced delay, as depicted in Fig. 1b<sub>1</sub>. With this additionally available slack, we are able to perform voltage scaling on more processes, thus the minimum energy schedule is 49.68% of  $E_{\mathcal{A}}^0$ . However, for such a voltage scaling the reliability is reduced to  $0.999\,999\,800 < R_g$ , and the reliability goal is missed.

Another approach, depicted in Fig. 1c, is not to mask fault occurrences at all. In this case, even the processes on different processors will be affected by the fault occurrence on the current processor. For example, an error in  $P_1$  on  $N_1$  will have to be communicated to processor  $N_2$  in order to switch to an alternative schedule that delays the scheduling of  $P_3$  which receives message  $m_1$  from  $P_1$ . This would create even more slack, leading to the schedule depicted in Fig. 1c<sub>2</sub>, that consumes only 33.96%  $E_{\mathcal{A}}^0$ , which is the largest obtainable energy reduction that does not violate the deadline. However, in this case the reliability of the system is further reduced to  $0.999\,999\,787 < R_g$ .

As the previous examples have shown, voltage scaling reduces reliability. The aim of this paper is to propose a schedule synthesis approach that is able to minimize the energy while at the same meeting the reliability and timing requirements. We will base our solution on the *transparent recovery* approach (Fig. 1b), which has been shown to quickly produce good quality schedules [11]. Thus, the scheduling algorithm is responsible for deriving offline the *root* schedules, i.e., the schedules in the no-fault scenario. The scheduler in each node, starting from the root schedule, based on the occurrence of faults, is able to derive *online*, in linear time, the necessary contingency schedule [11].

## 5.2 Motivational Example

Let us consider the example in Fig. 2 where we have an application of six processes mapped on an architecture of two nodes. All the relevant information is presented in the figure, similar to the previous example. Using transparent recovery, the shortest schedule without voltage scaling, is shown in Fig. 2a. The energy is  $E_{\mathcal{A}}^0$ , and the deadline of 260 ms and the reliability goal of 0.999 999 900 are met.

Optimizing the application for minimum energy consumption, with the deadline as a hard constraint, results in the schedule shown in Fig. 2b, where the energy consumed is 67.68% of  $E_{\mathcal{A}}^0$ . Process  $P_1$  is running at voltage level of 33% of  $V_{max}$ ,  $P_3$ ,  $P_4$  and  $P_6$  at 66%, while  $P_2$  and  $P_5$  are not scaled. In this case, the reliability is lowered to 0.999 999 878 which does not meet the reliability goal.

However, by carefully deciding on which processes are scaled and

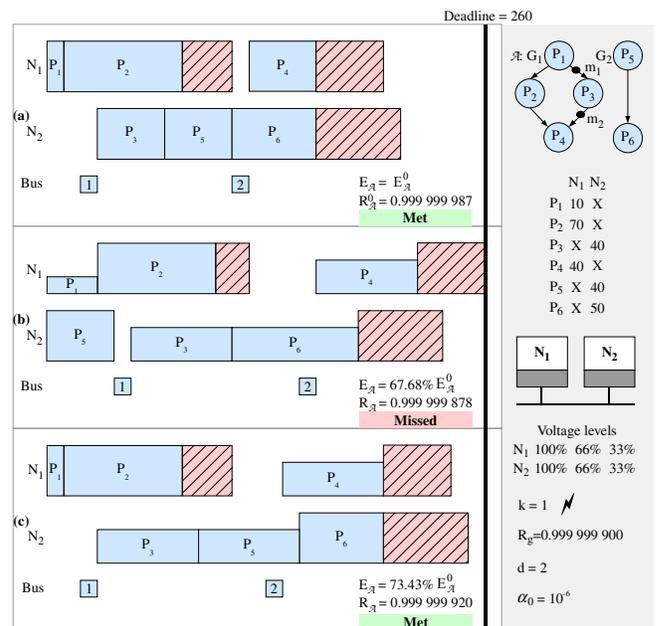


Figure 2. Scheduling and Voltage Scaling Example

by how much, it is possible to reduce the negative impact on reliability without a significant loss of energy savings. Thus, in Fig. 2c, by choosing not to scale  $P_1$ , and starting  $P_3$  before  $P_5$  on  $N_2$ , we are able to reach a reliability of 0.999 999 920, which meets the reliability goal at an only 5.75% reduction in energy savings compared to the minimum energy schedule in Fig. 2b.

This example shows that reliability has to be considered at the same time with scheduling and voltage scaling. Our CLP-based schedule synthesis strategy is able to produce schedules with constrained reliability, which yield, as the experiments will show, energy savings comparable to schedules with unconstrained reliability.

## 6. CLP-BASED SYNTHESIS STRATEGY

The problem presented in the previous section is NP-complete (scheduling in even simpler contexts is NP-complete [21]). In this section we present a constraint logic programming (CLP) approach for solving the problem. Thus, a system is 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.

The logic constraints used to model our problem fall under the following categories: (i) precedence constraints, (ii) resource constraints, (iii) timing, (iv) reliability and energy constraints, and (v) constraints for fault tolerance. Constraints (i)–(iii) have been extensively discussed in the literature [17]. The reliability and energy constraints (v) are captured by the equations introduced in the previous sections. Here we will concentrate on the constraints for fault tolerance.

### 6.1 Constraints for Fault Tolerance

When scheduling with fault tolerance using the transparent recovery technique, the precedence constraints have to take into account the recovery slack. There are two cases, treated separately.

1. *Processes on the same node.* Processes executed on the same processor share recovery slack. This slack is scheduled immediately after the processes, and thus will not impact the precedence constraints. Such a situation is depicted in Fig. 1b<sub>1</sub>, where  $P_1$  and  $P_2$  are mapped on the same processor, and share recovery slack. Thus, the constraint for processes on the same processing element is simply  $\mathcal{M}(P_i) = \mathcal{M}(P_j)$ .
2. *Processes on different nodes.* Things are more complex if the two processes are mapped on different processors. In such a situation, a 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. These are illustrated in Fig. 3a and b, respectively.

Let us consider the dependency between processes  $P_2$  and  $P_3$ . In Fig. 3a,  $P_2$  is scheduled after  $P_1$ . The figure shows the critical recovery path. This is the path which determines when data is available to be transmitted. In this example the longest recovery path is  $k$  re-executions of  $P_2$ , and hence  $P_3$  can start at time:  $Start(P_3) \geq Start(P_2) +$

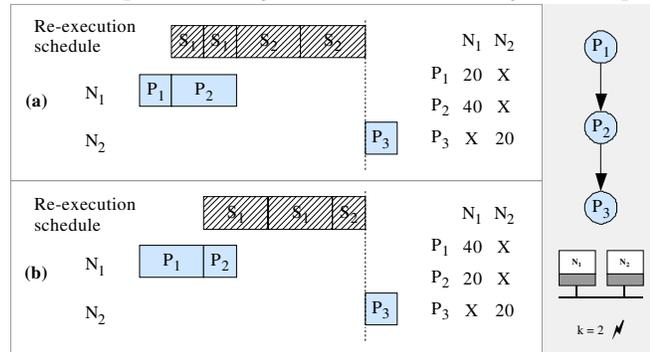


Figure 3. Fault-tolerance Constraints Example

$\lceil C_2 / f_2 \rceil + k \times C_2$ . In Fig. 3b,  $P_2$  is now scheduled after  $P_1$  ( $P_1$  has a larger execution time in this example). In this case, the longest recovery path to  $P_3$  is  $k$  re-executions of  $P_1$  plus a single execution of  $P_2$ . That is, the availability of data is not only determined by the sending process, but also the processes scheduled before it. The start time of  $P_3$  is constrained by:  $Start(P_3) \geq Start(P_1) + \lceil C_1 / f_1 \rceil + k \times C_1 + C_2$ .

These two schedule examples show that the availability of data does not only depend on the two processes which communicate, but also on all the processes with which the sending process shares slack. To generalize the shown constraints, in a way that can be used in the CLP model, detailed information of the recovery schedule is needed. This is achieved by creating a separate schedule for the recovery processes. For the examples shown in Fig. 3, the created recovery schedule is depicted in the Gantt chart entitled “re-execution schedule”.

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 precedence and resource constraints are imposed. The finishing times of the processes in the recovery schedule are described by:

$$Finish(S_i) \geq Start(P_i) + \lceil C_i / f_i \rceil + k \times C_i \wedge \quad (7)$$

$$Finish(S_i) \geq Start(S_i) + C_i$$

Note that the first part of the expression, up to the “and” operator, captures the situation depicted in Fig. 3a. The rest relates to Fig. 3b.

Using the recovery schedule, the general logic constraint for processes on different processors can now be written:  $Start(P_i) \geq Finish(S_i)$ . With the previous definitions of the recovery schedules and constraints for processes on the same, and on different processors, a general constraint for slack sharing can be derived:

$$\mathcal{M}(P_i) = \mathcal{M}(P_j) \wedge Start(P_j) \geq Start(P_i) + \lceil C_i / f_i \rceil \vee \quad (8)$$

$$Start(P_j) \geq Finish(S_i)$$

In the last part of the expression it is not stated that  $\mathcal{M}(P_i) \neq \mathcal{M}(P_j)$ , as this is an implicit consequence of the first part of the clause.

## 7. EXPERIMENTAL RESULTS

For the evaluation of our techniques we used applications of 10, 15, 20, 25 and 30 processes mapped on architectures consisting of 3 nodes. Ten examples were randomly generated for each application dimension, thus a total of 40 applications were used for experimental evaluation. Execution times were assigned randomly within the 10 to 100 ms. We have ignored communications for the experiments. The failure rate constant has been set to  $d = 2$  and the initial failure rate  $\alpha_0 = 10^{-6}$  faults per second. Half of the processes in the graphs have been randomly chosen to be made redundant using re-execution. The remainder of the processes are considered non-critical, and are not made redundant. We have used the ECL<sup>PS</sup> CLP system [1], version 5.10\_44 on 3.5 GHz AMD 64-bit computers with two gigabytes of RAM. We have set a progressive timeout for each run, based on the application size, to 10, 15, 20, 25 and 30 minutes, respectively. The best result determined during each run has been used in the evaluations.

In the experiments, the fully transparent schedule without voltage scaling has been used as reference. This is a schedule that a designer would determine in a straightforward manner, by introducing an amount of  $k \times C_i$  recovery slack after each process  $P_i$ . Let us call this approach *Straightforward Solution*, or SS. The deadline for the graphs in the experiments has been set to the length of the optimal SS schedule. The reliability goal has been set to:  $R_g = 1 - 10(1 - R_A^0)$ , which means that the probability of faults happening may be no more than ten times greater than in the schedule without voltage scaling. We have considered two situations, with  $k = 1$  and  $k = 2$ .

We were first interested to determine the impact of voltage scaling on reliability. For this, we have applied our scheduling and voltage scaling optimization with the objective of minimizing the energy

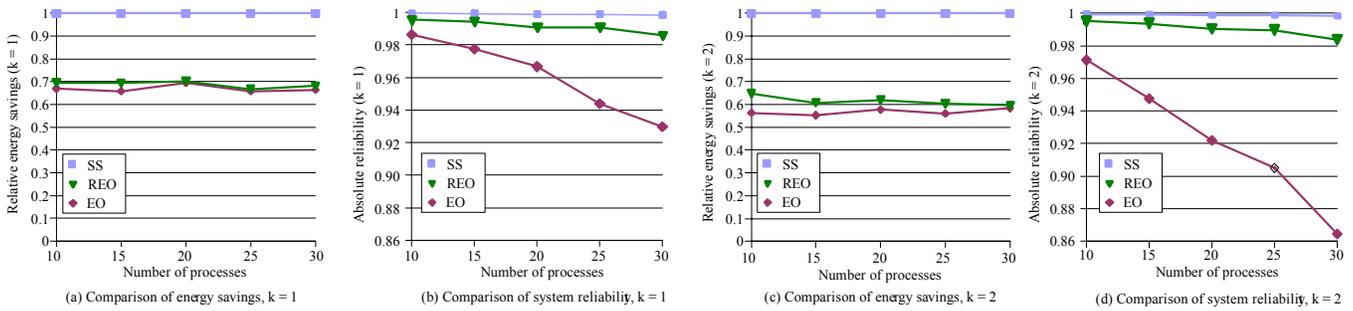


Figure 4. Experimental Results

consumption, but without imposing any reliability constraints. Let us denote this approach with *Energy Optimization*, EO. The results, averaged over the 10 applications for each application size, are presented in Fig. 4 for  $k = 1$  and 2. In the energy plots, Fig. 4a and c, we present the energy consumption obtained with EO relative to that of SS as the application size increases. The reliability plots in Fig. 4b and d present the absolute reliability of the approaches evaluated. As we can see from the figures, EO is able to obtain close to 40% energy savings compared to SS. These savings increase as  $k$  increases. However, this is at the expense of a significant reduction in reliability, which drops rapidly as the application size and  $k$  increase. For example, for an application of 30 processes and  $k = 2$ , the average reliability is below 0.87. Therefore, the reliability has to be addressed during scheduling and voltage scaling.

This was the focus of our second round of experiments. We have performed scheduling and voltage scaling with the goal of minimizing the energy as in EO, but we have additionally imposed the reliability constraint that the resulted system reliability has to be within the reliability goal  $R_g$  (as set by Equation ). We have called this approach *Reliable Energy Optimization* (REO). As we can see in Fig. 4b and d, the reliability with REO no longer drops below  $R_g$ . Moreover, as Fig. 4a and c show, the energy savings of REO relative to SS are comparable to those of EO, which does not care about reliability. This means that our CLP-based scheduling and voltage scaling approach is able to produce reliable implementations without sacrificing the reduction in energy consumption.

We have also considered an MP3 encoder application [20]. The deadline for the application is 25 ms. The MP3 is executed on an architecture with two processing elements that can be run at three voltage levels of 100%, 70% and 50% of  $V_{max} = 3V$ . For  $k = 1$  and  $R_g = 0.999\ 999\ 999$ , the unconstrained-reliability schedule determined by EO consumes 53.2% of  $E_a^0$ . The reliability goal is missed, since the resulted reliability is 0.999 999 996. However, by using REO we have made the designed system meet its reliability goal, sacrificing only 9% of the energy savings.

## 8. CONCLUSIONS

In this paper we have addressed the scheduling and voltage scaling for fault-tolerant hard real-time applications mapped on distributed embedded systems where processes and messages are statically scheduled.

We have captured the effect of voltage scaling on system reliability and we have shown that if the voltage is lowered to reduce energy consumption, the reliability is significantly reduced. Hence, we have proposed a CLP-based approach that takes reliability into account when performing scheduling and voltage scaling.

As the experimental results have shown, our CLP-based strategy is able to produce energy-efficient implementations which are schedulable and fault tolerant. By carefully deciding on the start times and voltages of processes we have shown that it is possible to eliminate the negative impact of voltage scaling on reliability without a significant loss of energy savings.

## REFERENCES

- [1] K. Apt, M. Wallace, *Constraint Logic Programming using ECL<sup>i</sup>PS<sup>e</sup>*, Cambridge University Press, 2006.
- [2] A. Bertossi, L. Mancini, "Scheduling Algorithms for Fault-Tolerance in Hard-Real Time Systems", *Real Time Systems*, 7(3), 229–256, 1994.
- [3] A. Burns et al., "Feasibility Analysis for Fault-Tolerant Real-Time Task Sets", *Euromicro Workshop on Real-Time Systems*, 29–33, 1996.
- [4] Y. Cai et al., "Cache Size Selection for Performance, Energy and Reliability of Time-Constrained Systems", in *Proc. Asia and S. Pacific Design Autom. Conf.*, 923–928, 2006.
- [5] A. P. Chandrakasan, S. Sheng, R. W. Brodersen, "Low-power CMOS digital design", in *IEEE Journal of Solid-State Circuits*, 27(4), 473–484, 1992.
- [6] V. Claesson, S. Poldena, J. Söderberg, "The XBW Model for Dependable Real-Time Systems", *Parallel and Distributed Systems Conf.*, 1998.
- [7] C. Dima et al., "Off-line Real-Time Fault-Tolerant Scheduling", in *Proc. of Euromicro Parallel and Distributed Processing Workshop*, 410–417, 2001.
- [8] A. Ejlali, B. M. Al-Hashimi, M. Schmitz, P. Rosinger, S. G. Miremadi, "Combined Time and Information Redundancy for SEU-Tolerance in Energy-Efficient Real-Time Systems", in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(4), 323–335, 2006.
- [9] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems", *IEEE Real-Time Systems Symposium*, 152–161, 1995.
- [10] G. Fohler, "Adaptive Fault-Tolerance with Statically Scheduled Real-Time Systems", in *Euromicro Real-Time Systems Workshop*, 161–167, 1997.
- [11] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems", in *Proc. of Design Automation and Test in Europe Conf.*, 864–869, 2005.
- [12] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems", in *Proc. of DATE*, 706–711, 2006.
- [13] B. W. Johnson, *The Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.
- [14] N. Kandasamy, J. P. Hayes, B. T. Murray, "Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems", in *IEEE Transactions on Computers*, 52(2), 113–125, 2003.
- [15] H. Kopetz, *Real-Time Systems—Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.
- [16] H. Kopetz et al., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", in *IEEE Micro*, 9(1), 25–40, 1989.
- [17] K. Kuchcinski, "Constraints-Driven Scheduling and Resource Assignment", in *ACM Trans. on Design Autom. of Electronic Syst.*, 8(3), 355–383, 2003.
- [18] R. Melhem, D. Mosse, E. Elnozahy, "The interplay of power management and fault recovery in real-time systems", in *IEEE Transactions on Computers*, 53(2), 217–231, 2004.
- [19] C. Pinello, L. P. Carloni, A. L. Sangiovanni-Vincentelli, "Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications", *DATE Conf.*, 1164–1169, 2004.
- [20] M. T. Schmitz, B. M. Al-Hashimi, P. Eles, *System-Level Design Techniques for Energy-Efficient Embedded Systems*, Springer, 2003.
- [21] D. Ullman, "NP-Complete Scheduling Problems," in *Journal of Computer Systems Science*, vol. 10, 384–393, 1975.
- [22] Y. Zhang and K. Chakrabarty, "Dynamic adaptation for fault tolerance and power management in embedded real-time systems," in *ACM Transactions on Embedded Computing Systems*, vol. 3, 336–360, 2004.
- [23] D. Zhu, R. Melhem and D. Mossé, "The Effects of Energy Management on Reliability in Real-Time Embedded Systems," in *Proc. of the International Conference on Computer Aided Design* 35–40, 2004.
- [24] D. Zhu and H. Aydin, "Reliability-Aware Energy Management for Periodic Real-Time Tasks", in *Proc. of the Real-Time and Embedded Technology and Applications Symposium*, 225–235, 2007.