Reliability-Aware Energy Optimisation for Fault-Tolerant Embedded MP-SoCs

Kåre Harbo Poulsen
s001873

Supervisor: Paul Pop

Kongens Lyngby
DTU ◇ IMM
March 19, 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Embedded computing systems are making their way into more and more devices, from household appliances to mobile phones, and from PDAs to cars. Many of these systems are battery powered, and hence battery lifetime is a critical design issue. Also these systems, need to meet the timing constraints imposed by their application domain.

An increasing number of application areas for real-time embedded systems, such as space and consumer applications, have hard constraints both in terms of energy and reliability. To address these two simultaneously is challenging because lowering the voltage to reduce power consumption, which is the most common approach, has been shown to exponentially increase the number of transient faults. Moreover, time-redundancy based fault-tolerance techniques, such as re-execution, and voltage scaling-based low-power techniques are both relying on the use of processor idle-time.

In addition, such competing requirements have to be met within a given development and manufacturing cost and time-frame. Therefore, the task of designing such embedded systems is becoming not only increasingly important, but also increasingly difficult. The objective of this thesis is to develop techniques which are able to simultaneously meet both energy and reliability constraints at system-level.

In this thesis real-time applications with hard deadlines, mapped on distributed multi-processor systems-on-a-chip, are considered. The applications are represented as a set of interacting processes and have hard reliability and timing requirements. Processes and messages are statically scheduled using schedule tables. I propose techniques for the scheduling, mapping, voltage scaling and redundancy assignment, such that the energy consumption of the applications is minimised, and the implementations are schedulable and meet the imposed reliability goals.

The techniques have been implemented using a constraint logic programming system, and have been evaluated using a set of synthetic applications, as well as a real-life application, consisting of an *MP3-decoder*. The experiments show that, using careful optimisation, it is possible to produce reliable and schedulable implementations without compromising energy consumption.

Resumé

Indlejrede systemer bliver mere og mere almindelige i disse år. Både i applikationer som mobiltelefoner og *PDA*er, men også i hjemmets maskiner. Disse systemer er ofte batteridrevne, og det er derfor nødvendigt, at de sparer på strømmen. Systemernes funktioner stiller desuden krav til, at de kan operere i real-time.

Et stigende antal anvendelsesområder for indlejrede systemer har tydelige begrænsninger både inden for energiforbrug og pålidelighed. Den mest almindelige fremgangsmåde til at sænke energiforbruget er en dynamisk nedsættelse af spændingen. Men dette giver anledning til en eksponentiel stigning i antallet af fejl, hvilket gør det besværligt at fremstille systemer med høj pålidelighed og lavt energiforbrug. Energibesparende teknikker konkurrerer desuden med teknikkerne til fejltolerance om at gøre brug af systemets *slack*, dvs. den tid hvor systemet ikke udfører opgaver.

Disse konkurrerende krav skal opfyldes inden for et firmas tidsplan samt produktions- og fremstillingsomkostninger. Dermed bliver det at designe indlejrede systemer, der både har et lavt energiforbrug og en høj pålidelighed, ikke blot en mere vigtig, men også en mere vanskelig opgave. Formålet med dette arbejde er at udvikle teknikker, der både kan opfylde energikrav og pålidelighedskrav i systemets designfase.

Dette arbejde undersøger real-time-applikationer med strenge tidsbegrænsninger. Disse er allokeret på distribuerede multiprocessor *system-on-a-chip*-systemer. Applikationerne repræsenteres som et sæt af kommunikerende processer, der har strenge begrænsninger for både pålidelighed og timing. Processernes starttider og kommunikation er statisk fastlagt. I dette arbejde præsenteres en række teknikker, der indfører fejltolerance og som samtidig bestemmer starttider, hardwareallokation og spændingsregulering. På denne måde kan et systems energiforbrug blive minimeret, samtidig med at et pålidelighedsmål bliver opfyldt.

De fremstillede teknikker er blevet implementeret i et *constraint logic programming* system og er blevet evalueret ved hjælp af syntetiske applikationer. Dertil kommer også en virkelig applikation i form af en *MP3*-dekoder. De udførte eksperimenter viser, at det ved hjælp af god optimering er muligt at opnå systemer, der både har en høj pålidelighed og et lavt energiforbrug.

Contents

Chapter 1

Introduction

The tendency has for a long time been for digital systems to make their way into more and more everyday appliances. Both in highly advanced devices such as mobile phones, *mp3* players, *PDA*'s and other portable devices, but also cars, and even low tech devices such as household appliances.

These embedded systems are often battery powered, and hence need to have low power consumption, in order to yield good battery lifetime. They need to be high performance, to meet the timing constraints imposed on the device (e.g. real time voice coding/decoding in mobile phones, music decoding in music players etc.). In addition, the devices need to be small, as they are intended to be part of the users everyday equipment, or even act as an accessory to express style or interests. These devices need to be reliable, as their functionality is often relied upon, as in mobile phones, or may even be safety-critical as in the safety systems of cars.

The rest of this chapter is organised as follows. Section 1.1 introduces the design flow for embedded systems. The motivation for this project is presented in section 1.2. Section 1.3 presents the work by others relevant to this thesis, and the problem formulation and the contributions made in this thesis are presented in section 1.4.

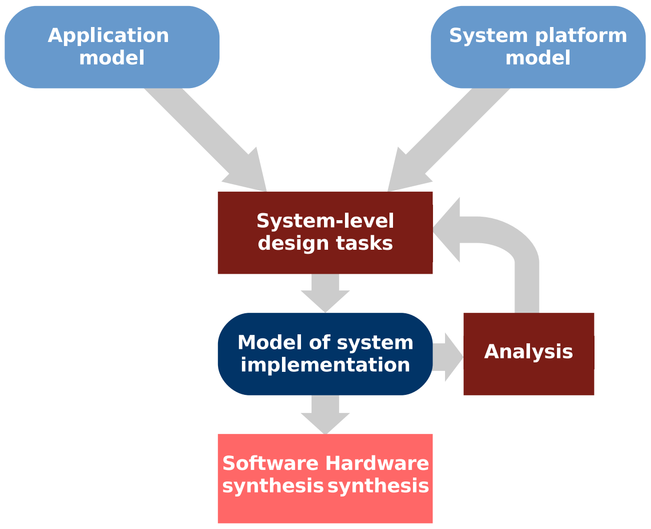


Figure 1.1: Design flow for embedded systems [23].

1.1 Embedded Systems Design Flow

Embedded systems are single purpose systems, with a well defined functionality. The system-level design flow for embedded systems is shown in figure 1.1. The flow has two inputs, namely: a model of the application, and a model of the hardware architecture on which the application is to be run. In this thesis the architectures considered are multiprocessor systems-on-a-chip (*MP-SoC*s), consisting of several processing elements interconnected by a bus.

Several design tasks are performed as part of the system-level design task. This includes assigning each part of the application onto a specific hardware unit on which it will run. This design task is called mapping. Further, a time plan, or schedule, has to be generated which dictates when the different parts of the application should be executed. This has to take into account data dependencies and deadlines, to ensure correct behaviour of the system. If the system is to use voltage scaling, a voltage schedule will also be derived at this stage.

To verify that the system description arrived at in the design phase is actually going to work correctly, a model is created of the system. Using this model the behaviour of the system is analysed to ensure that timing and power requirements are met. Typically the design phase will be an iterative process, where the design is gradually refined as part of an optimisation.

When a satisfactory system implementation has been found, the system is synthesised. This process creates the actual hardware and software implementations.

In this thesis I address hard real-time applications modelled using process graphs [17]. The functionality is distributed on a heterogeneous system of processing elements, interconnected by a bus. Processors and messages are statically scheduled using schedule tables.

In this work the focus will be on the system-level design tasks of scheduling, mapping and redundancy assignment.

1.2 Motivation

Traditionally, embedded systems have been designed by a number of single purpose chips assembled on a print-board. To accommodate the need for smaller components and better performance, more and more functionalities are today being integrated on single chips. This allows for making complete solutions on a single chip, or *system-on-chip* solutions. These systems will often include several digital processors, for e.g. speech coding, radio coding, etc., and are hence commonly called *multi-processor system-on-a-chip* (*MP-SoC*).

The continued increase in integration and complexity of *MP-SoC*s is made possible by the on going increase in available space on a chip. This phenomenon is described by *Moore's Law* [27, 28] which conjectures that the amount of transistors that can be fit on a single chip doubles every 18 months [25]. This law is continuously upheld, as new technologies are developed which allows for decreasing the size of single transistors. The reduced feature sizes, lead to an increase in power consumption. A prediction of the power consumption of processors as a function of time, is shown in figure 1.2. This increase in power consumption, combined with the increasing miniaturisation of features, lead to increased energy density. The energy density of future integrated circuit technologies will approach that of a nuclear power plant [8].

The increased miniaturisation also gives rise to another phenomenon, namely increase in the amount of faults. Faults in electronics are random and non-permanent electrical events, which are seen as bit flips in logics or memory. Faults due to internal reasons, such as leak current or cross talk, are called intermittent faults. Faults caused by external effects are called transient faults. External effects may be caused by electromagnetic radiation from other devices, or exposure to the ever present cosmic radiation. The latter is especially im-

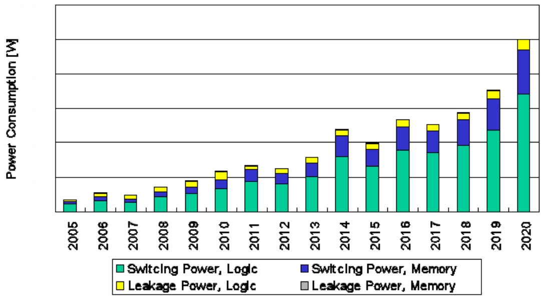


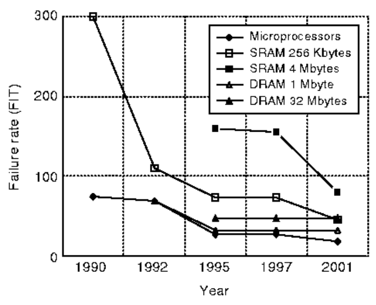
Figure 1.2: Prediction of power consumption for micro electronics (from [28]).

portant in space applications where the unshielded radiation can give rise to as many as 35 faults in 15 minutes [20], but is also an important factor in earth bound applications.In this thesis, I address transient faults, and do not dwell with their cause, but rather how to handle and recover from them.

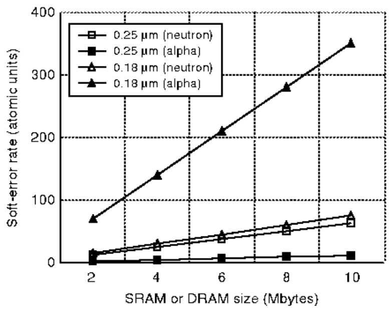
The failure rates for modern electronics are plotted in figure 1.3. The left plot shows that the amount of permanent faults is falling. However the number of transient faults are increasing rapidly. The shown plot for transient faults refers to memory units, but also applies to general logic circuitry.

The increase in energy consumption is often addressed by the use of energy management techniques. One very common approach is dynamic voltage scaling (DVS). This has been shown to be an easy and effective means of conserving power, but has also been shown to further increase the probability of faults [30, 32]. As a consequence of this effect, and the generally increasing probability of faults, it is becoming critical to consider faults in a system already in the design phase.

Design tools exist for embedded systems that can create system level designs. These allow for doing optimisation on different parameters, such as energy consumption, or fault tolerance. As shown in [30] these two tasks are not independent, but in fact greatly interact. Current design tools do not take this interaction into account, which may lead to them creating systems which are energy efficient, but very unreliable.



(a) Permanent-failure rate for CMOS devices.



(b) Transient-failure rate for CMOS memories.

Figure 1.3: Failure rate plots, for permanent faults and transient (soft) faults (from [3]).

1.3 Related Work

Several hardware solutions for fault tolerance have been proposed, e.g. MARS [14], TTA [13], and XBW [2], all of which use hardware redundancy to tolerate one permanent fault. These approaches are also able to tolerate transient faults, but they are very costly in terms of hardware. This cost is only further increased if the systems are to tolerate larger number of faults, a point that is increasingly important as the amount of transient faults is much larger than permanent faults [3].

Current research use cost as the only design constraint [16]. The use of redundancy, however, introduces overhead, in terms of performance, and thus may lead to systems that are unschedulable. Only few researchers [12, 21, 22] optimise their implementations to minimise the penalty on performance. For these, the optimisation is limited though, and does not consider the use of several redundancy techniques.

Two system-level approaches that allow an energy/performance trade-off during run-time of the application are dynamic voltage scaling (*DVS*) and adaptive body biasing (*ABB*) [24]. While *DVS*aims to reduce the dynamic power consumption by scaling down operational frequency and circuit supply voltage, *ABB* is effective in reducing the leakage power by scaling down frequency and increasing the threshold voltage through body biasing.

The current research has addressed fault-tolerance and low-power requirements separately. However, embedded systems using *DVS*and *ABB*, are more susceptible to transient faults, as the rate of these increase exponentially as the supply voltage decreases [32]. Conversely, increased voltage levels lead to higher on-chip temperatures, which in turn has a negative effect on reliability. Further, the energy management techniques, and time-redundant fault tolerance techniques, are competing for the same slack (unused time in schedules for processors). Initial research into the interplay of energy/performance trade-offs and fault-tolerance techniques has been presented in [6, 20, 30]. These approaches are very restricted in terms of situations considered, and are thus of limited interest.

1.4 Thesis Objective and Contributions

In this thesis hard real-time applications mapped on distributed multi-processor systems-on-a-chip are considered. The applications are represented as a set of interacting processes and have hard reliability and timing requirements. Processes and messages are statically scheduled using schedule tables. The objective of this thesis is to propose techniques for the scheduling, mapping, voltage scaling and performing redundancy assignment, such that the energy consumption of the applications is minimised, and the implementations are schedulable and meet the imposed reliability goal.

The techniques have been implemented using a constraint logic programming system, and have been evaluated using synthetic applications as well as a real-life example consisting of an *MP3-decoder*. The experiments show that, through careful optimisation, it is possible to obtain reliable and schedulable implementations without compromising the energy consumption.

The contributions of the thesis are the following:

- Design optimisation for energy minimisation under reliability and timing constraints.

  Energy minimisation is usually done using voltage scaling. However, research has shown that lowering the voltage will dramatically decrease reliability. Thus, if the reliability of a system is increased, by introducing redundancy, and then voltage scaled (within the deadlines), the reliability is destroyed. If a minimal-energy system is obtained, and redundancy then introduced, it might not meet the deadlines. The most important contribution of the thesis is a design optimisation method, which is able to produce reliable implementations, that minimise energy at the same time as meeting the deadlines.

- An optimisation method that decides the type of redundancy.

  To increase the reliability of a system, redundancy techniques such as re-execution and replication are needed. It is shown that using just re-execution is not enough, because both re-execution and voltage scaling compete for the slack. Using passive replication in conjunction with re-execution, can better exploit the slack. This is because, if slack is not available on one processor, it might be found on another processor.

- A *constraint logic programming*-based scheduling technique which is able to quickly produce good quality schedules.

  Having a good scheduling algorithm can help in increasing the slack. With increased slack, the reliability-energy trade-offs can better be supported.

Chapter 2

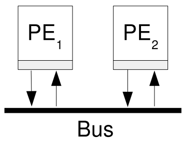
Preliminaries

In this section I present the preliminaries for the work in this thesis. Section 2.1 presents the fundamentals of mapping and scheduling. The system and application models are presented in section 2.2 and 2.3 respectively. In section 2.4 the fault model is introduced, and section 2.5 introduces fault recovery. An alternative application model with explicit fault recovery is presented in section 2.6. In section 2.7 the concept of reliability is introduced, and equations are presented. The models for energy and reliability under voltage scaling are presented in sections 2.8 and 2.9. The software model, and the corresponding scheduler implementations are presented in section 2.10.

2.1 Scheduling and Mapping

As mentioned in the introduction, mapping and scheduling are the design tasks of assigning processes to hardware units, and making a time plan for the execution of the processes. Both of these problems are individually *NP*-complete. Consequently the combination of the two is also *NP*-complete [7, 29].

This makes the problems computationally hard, and hence sophisticated algorithms are needed to solve these. The work in this thesis uses *constraint*



(a) Architecture

	Voltage Level		
PE_1	100 %	67 %	34 %
PE_2	100 %	67 %	34 %

(b) Voltage levels

Figure 2.1: Sample architecture with two processing elements each with three frequency levels.

logic programming (*CLP*) to model the problems, and achieve optimal solutions. Finding optimal solutions is generally not feasible using conventional programming.

2.2 System Model

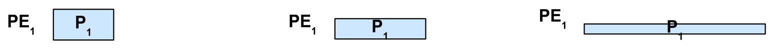
In this thesis a system model, consisting of a number of processing elements PEs that are connected by a single bus, is considered. These processing elements may be heterogenous and have different performance, and thus take different amounts of time to execute the same process. A processing element can be run at a number of preset frequency levels $\mathcal{F}_{PE}$. These frequency levels are expressed in percent of the processors maximum performance.

A sample architecture is shown in figure 2.1, where both the layout of the processing elements and the available frequency levels for each processing element are shown.

Each processing element has a real-time operating system, which is responsible for starting processes. Processes are started in accordance with a pre-rendered static schedule table, or a set of schedule tables (as discussed later). The operating system monitors whether processes execute successfully, and if not, takes measures to tolerate the fault in accordance with the fault tolerance policy.

2.3 Application Model

An application $\mathcal{A}$ is modelled as a directed acyclic graph. A graph $\mathcal{G}$ consists of a set of edges $\mathcal{E}$ and vertices $\mathcal{V}$ such that $\mathcal{G}(\mathcal{V}, \mathcal{E}) \in \mathcal{A}$. Each vertex represents a process P_i with a corresponding worst-case execution time (c). Since we



(a) $\mathcal{F}_{P_1} = 100\%$, c= 2 (b) $\mathcal{F}_{P_1} = 67\%$, c= 3 (c) $\mathcal{F}_{P_1} = 34\%$, c= 6

Figure 2.2: Voltage scaling of a single process onto PE_1 from figure 2.1. The height of the process illustrates the frequency its run at, and the length the duration of the process.

operate with a heterogene architecture c is specified per processing element, as it will be a function of the processing elements design and performance. The communication between, and thereby ordering of, processes, are represented by the edges. An edge $e_{ij} \in \mathcal{E}$ denotes a communication from process P_i to P_j.

Figure 2.3 shows a sample application, with its process graph and the corresponding worst case execution times for the architecture in figure 2.1.

The specified c for a process P_i corresponds to the execution time for the process run at $\mathcal{F}_{\mathcal{P}_\rangle} = 100\%$. The execution time c_f for a process run at a lower frequency f is given by ([31]):

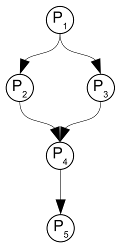
$$c_f = \frac{c_0}{f} \tag{2.1}$$

Figure 2.2 shows a process scheduled on the same processor, but at three different frequency levels. To visually capture that the frequency is lowered in the Gantt chart, the height of the process is decreased. The length of the process shows how the c of the process increases as the frequency is lowered.

For the fault tolerance techniques described in the following, it is considered that the worst case execution time of a process includes the time needed to do error detection, so the c is the sum of the time to do a failed execution, detect it and clean up and set up for recovery execution. This allows us to disregard which error detection method is used, as this is outside the scope of this thesis, and this subject is well researched in the works of others (for insight on this subject the reader is directed towards [11] and [25]).

2.4 Fault Model

A system may experience different kinds of faults during its execution. It may either be permanent faults, or transient or intermittent faults. The work in this



(a) Process graph $\mathcal{G}$

Process	PE_1	PE_2
P_1	2	2
P_2	2	2
P_3	2	2
P_4	2	2
P_5	2	2

(b) Durations table

Figure 2.3: Sample application $\mathcal{A}$

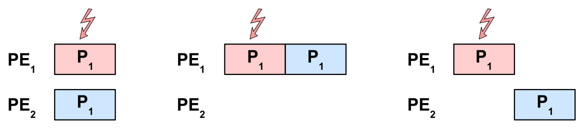
thesis only deals with non-permanent faults, as these are much more frequent than permanent faults.

The arrivals of faults can be modelled by a Poisson distribution with an arrival rate λ , referred to as the *failure rate* [11]. For a single chip system reasonable values for the failure rate are in the range $10^{-8} - 10^{-6}$ per second [32]. This is equivalent to 100.000 FITs, i.e. failures in time, or failures per billion hours of use per megabit. The designer of a system will impose a minimum reliability, a reliability goal R_g, based on the reliability requirements of the application. Based on λ and R_g, the number of transient faults that will be tolerated k is determined. In the literature of fault tolerance, reliability goals are often stated in terms of the number of nines after the zero. For instance the goal $R_g = 0.9999991$ is called *6 nines (and a 1)*. This terminology is adopted in this thesis, when numerical values of reliabilities are discussed. In order to meet this goal, the number of faults k to be tolerated by the system is determined.

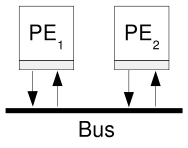
Within a single execution of an application the distribution of faults is random, and may strike any process. For $k > 1$ any combination of processes or even the same process may be struck k times.

2.5 Fault Recovery

To recover from a failed process it is necessary to add redundancy. This redundancy can be spatial, i.e. the process is run simultaneously on different processing elements, this is called replication. Alternatively the process can be made temporally redundant, i.e. the process is redundant in time, and is scheduled after the failing process on the same processing element, this is called



(a) Replication (b) Re-execution (c) Passive replication



(d) Architecture

Process	PE_1	PE_2
P_1	2	2

(e) Durations

Figure 2.4: Recovery techniques

re-execution. The latter technique has the advantage that the recovery run of the process is only executed in the event of a fault. This can be combined with replication, and is then called passive replication, where the process is scheduled after the failing process, but on another processing element. All the recovery-techniques are shown with examples in figure 2.4.

In this thesis, the first execution of a process is called the root process, and the following executions are called recovery processes. Similarly I use root schedule and recovery schedule.

Another commonly used fault tolerance technique used is checkpointing [20]. This technique can be modelled using re-execution, and is hence not covered specifically in this thesis.

To use the presented fault tolerance techniques it is critical that the system is able to detect faults. Fault detection is well covered in the literature. Common techniques include fingerprinting, where output bits are coded, and time-stamping where the execution of a process is timed, and is considered faulty if it does not finish within its c. How fault detection is done is outside the scope of this thesis, and is not covered further. The fault detection implemented by the designer is assumed to be sufficient to meet R_g. The interested reader is directed towards [11] and [25].

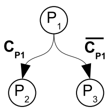


Figure 2.5: A sample conditional process graph. Process P_1 produces the condition C_{P_1}. If this is true P_2 will be executed, if it is false P_3 will be executed.

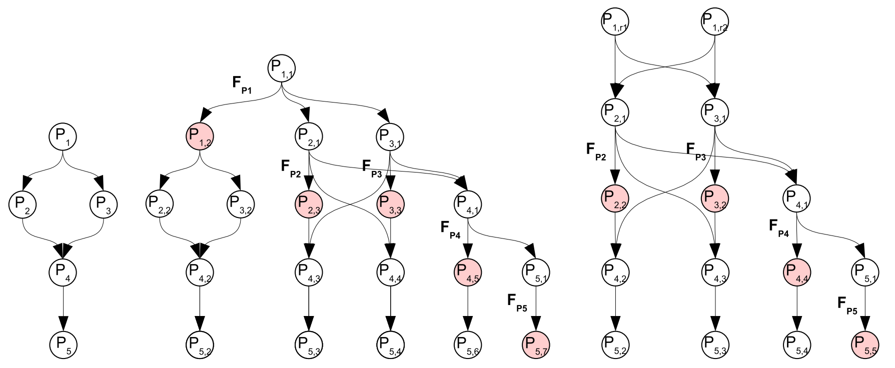
2.6 Fault-Tolerant Conditional Process Graphs

Conditional process graphs are an extension of normal process graphs, which adds the notion of guards, or conditions, on some edges. Conditions are boolean, and may be either *true* or *false*. A conditional process P_i, that produces the condition C_{P_i}, will have the conditional output edges e_{ij} which are guarded by the outcome of the condition.

Figure 2.5 shows a simple conditional process graph. The process P_1 produces the condition C_{P_1}. If this evaluates to *true* the edge guarded by this condition, marked C_{P_1} is chosen, and process P_2 is executed. If the condition is *false*, the edge marked $\overline{C_{P_1}}$ is chosen and P_3 is executed. These two paths are mutually exclusive, as they depend on different outcomes of the same condition.

In [9] and [10] conditional process graphs are extended to capture all possible execution scenarios in case of faults. Such a graph is called a *fault-tolerant conditional process graph* (*FT-CPG*). A process P_i produces a condition, corresponding to the success of its execution. If it fails it will have the condition F_{P_i}, and if it executes without faults $\overline{F_{P_i}}$. An example of a fault tolerant conditional process graph is shown in figure 2.6(b). For ease of reading, only edges which model faults are marked by the condition. Tinted processes are recovery executions. The shown *FT-CPG* captures all the fault scenarios depicted in the example in figure 2.10. For example the scenario captured in the schedule in figure 2.10(b) on page 27 is captured by the left-most branch in the *FT-CPG.*

Deriving an *FT-CPG* that captures all the fault scenarios of a process graph corresponding to k transient faults, is not trivial. In this section we shall not dwell further on this. An algorithm for deriving such graphs is presented in appendix A.



(a) Original process graph (b) *FT-CPG* for the graph, with $k = 1$ (c) *FT-CPG* for the graph, with $k = 1$ and P_1 replicated

Figure 2.6: Examples of a process graph, and its derived *FT-CPG* graphs for $k = 1$. Tinted processes mark recovery executions.

2.7 Reliability

The reliability of a system is a measure for the probability of its successful execution. In this section I present the reliability model used in this thesis, firstly for single processes. Secondly, I will use the formula for a single process to derive a general expression for the reliability of a fault tolerant application. Illustrations of the different recovery techniques are shown in figure 2.4.

2.7.1 Single Process Reliability

The reliability R_0 of a process is defined as the probability of its successful execution [11].

$$R_0 = e^{-\lambda c} = 1 - \rho \tag{2.2}$$

Where c is the execution time of the process, given by equation 2.1. ρ is the probability of failure. The term λ is the failure rate, which describes the amount of errors that will occur within a time unit.

2.7.2 Reliability of Re-Execution

For a system with the ability to handle k faults, a process will have k recovery executions scheduled after the root execution. For such a setup, the reliability is given by the probability of *not* all processes failing. Formally this is expressed as:

$$R_{P_{Reex}} = 1 - (1 - R)^{1+k} \tag{2.3}$$

Where the last term is the probability of all processes failing in the same run.

2.7.3 Reliability of Replication

Similarly, for a process scheduled to handle k faults by replication, the reliability is also given by the probability of *not* all processes failing, and is written as:

$$R_{P_{Repl}} = 1 - \prod_{i=1}^{k}(1 - R_i) \tag{2.4}$$

Where again the last term is the probability of all executions failing. The expression contains a reliability term for each execution of the process (in contrast to the formula for re-execution) as the processes are mapped to different processing elements, which may have different performance and reliability properties. If all processing elements are identical the reliability will simplify to equation (2.3).

This expression for replication also holds for passive replication.

2.7.4 Application Reliability

An application consists of a number of processes, for each of which the above equations yield the reliability. Since all of these processes must execute successfully, and I assume that the execution of each processes is *independent*, the reliability of an application $\mathcal{A}$ is:

$$R_{\mathcal{A}} = \prod_{P_i \in \mathcal{A}} R_{P_i} \tag{2.5}$$

Equipped with this equation and the presented general expressions for calculating reliabilities for single processes, the reliability for any fault tolerant application can be evaluated.

2.8 Power Model

Power in electronics is mainly consumed as dynamic power, i.e. the power that is needed to drive the internal bits from one value to the other. This is called active power. Active power depends greatly on the clock speed at which the circuitry is driven, as it is necessary to use more power to do faster switching. In contrast, passive power is the power that dissipates from the circuitry regardless of the running frequency.

As there is an almost linear relation between the frequency of a system, and the voltage needed to drive this [31], I shall be using the terms voltage scaling and frequency scaling interchangeably in the rest of the thesis.

In this thesis I use the power model from [32] which describes the consumed power as:

$$P = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{eff} f^m) \tag{2.6}$$

In which $\hbar$ is a boolean variable, which takes the value 1 if the system is powered up, and 0 if the system is in sleep mode. P_S is the passive power, which is always consumed by the circuit. P_{ind} is the frequency independent component of the active power. Finally, P_d is the frequency dependent component. The frequency dependent component is extended to be described as an effective capacitance C_{eff} and a frequency f^m, where m is the dynamic power exponent, an architecture dependent number, for which $m \geq 2$ [31].

In this thesis I assume that the *MP-SoC*s do not support switching to sleep mode, thus $\hbar$ will always be 1. As the work in this thesis focuses on the energy savings obtainable from using energy management techniques, the passive component of the power P_s can be disregarded as it will only contribute as a constant. In this way we arrive at:

$$P = P_{ind} + C_{eff} f^m \tag{2.7}$$

This gives the energy consumption for a process P_i [30]:

$$E_{P_i} = (P_{ind} + C_{eff} f_{P_i}^m) c_{P_i} \tag{2.8}$$

Where f_{P_i} is the frequency at which it is executed. Generalising this for a set of processes P in an application $\mathcal{A}$ for which $P_i \in \mathcal{A}$ we get the power for an application:

$$E_{\mathcal{A}} = \sum_{P_i \in \mathcal{A}} (P_{ind} + C_{eff} f_{P_i}^m) c_{P_i} \tag{2.9}$$

It should be noted that this is not a precise measure of the exact power consumed, as the passive components would then need to be part of the equations, but rather a means of comparing different design alternatives. The model allows for determining the possible energy savings, and as the aim of this work is to do just that, the model is appropriate.

Numerical examples of how to use the energy expressions are given in section 3.3.

Precise expressions for the power consumption of embedded systems are presented in [24].

2.9 Reliability with Voltage Scaling

Lowering the voltage minimises the energy. However, it has been shown that it also dramatically lowers the reliability [30].

In this section, voltage scaling is introduced into the reliability formulas from section 2.7, to capture how the reliability of voltage scaled systems decreases.

As described the failure rate of a system is dependent on the frequency level the system is run at. The relation between the two can be described by the expression proposed in [30, 32]:

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \tag{2.10}$$

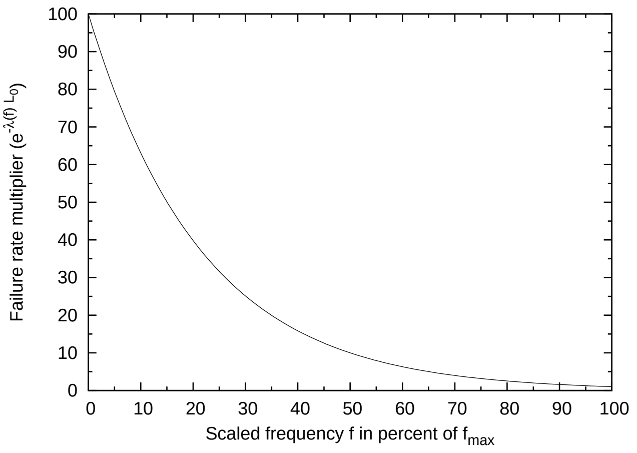


Figure 2.7: Plot of the relation between the failure rate multiplier and frequency introduced in equation (2.10). The normalised frequency ranges from 0, the minimum frequency of the processor ($f_{min} \geq 0$) to 1, the processor's f_{max}.

In which λ_0 is the failure rate of the processor when run at maximum frequency f_{max}, and d is an architecture specific constant. In figure 2.7 the frequency dependent λ is plotted for $\lambda_0 = 1$. The plot shows that the failure rate increase is moderate for frequency levels down to about 60%. However, for lower frequencies, the failure rate increases dramatically, and for a processor run at minimum frequency the failure rate will be 100 times greater.

In order to yield best possible reliability, recovery executions are always executed at full speed. Using the formulas presented in section 2.7, expressions for the reliability of processes with voltage scaling can now be deduced.

2.9.1 Single Process Reliability

Using equation (2.2), the reliability of a single process, run at frequency f is:

$$R_s = 1 - \rho_s = e^{-\lambda_s c_s} = e^{-\lambda(f) c_s} \tag{2.11}$$

2.9.2 Reliability of Re-Execution

As I now have the expressions for scaled and unscaled processes, I can derive the reliability of a process with fault tolerance. Lets consider a process for which re-execution provides tolerance for one fault. Bearing in mind that re-executions will always run at the maximum speed, the reliability of re-execution is:

$$R_{Reex} = 1 - (1 - R_0)(1 - R_s) = 1 - (1 - e^{-\lambda_0 c_0})(1 - e^{-\lambda(f)c_s}) \tag{2.12}$$

In this expression the first parenthesis is the probability of the recovery execution failing. The second is the probability of root execution failing. Together they form the probability of both failing in the same run. This expands into [30]:

$$R_{Reex} = e^{-\lambda(f)c_S} + (1 - e^{-\lambda(f)c_s})R_0 \tag{2.13}$$

Where c_s is the execution time of the voltage scaled process.

The generalised expression for a system handling k faults is:

$$R_{Reex} = 1 - (1 - R_0)^k(1 - R_s) = 1 - (1 - e^{-\lambda_0 c_0})^k(1 - e^{-\lambda(f)c_s}) \tag{2.14}$$

2.9.3 Reliability of Replication

Since replicated processes are executed at the same time, there are no recovery processes, that will be run at full speed afterwards, in case of an error. As a consequence, *all* replicas may be voltage scaled, and the reliability for replication is thus different from that for re-execution.

The reliability of a replicated process, is again the probability of *not* all executions failing. For a system that handles 1 fault by executing the same process on two processing elements, the reliability is:

$$R_{Rep} = 1 - \rho_{f,1}\rho_{f,2} = 1 - (1 - R_{f,1})(1 - R_{f,2}) = 1 - (1 - e^{-\lambda(f)_1 c_1})(1 - e^{-\lambda(f)_2 c_2}) \tag{2.15}$$

And generalised for a system that handles k faults by having $k+1$ replicas:

$$R_{Rep} = 1 - \prod_{i=1}^{k+1} (1 - e^{-\lambda(f)_i c_i}) \tag{2.16}$$

2.9.4 Reliability of Passive Replication

Passive replication is similar to replication in terms of all processes being arbitrarily mapped. But similar to re-execution in terms of all recovery executions being scheduled at full speed. Hence the expression for reliability of passive replication is a combination of the two:

$$R_{PRep} = 1 - (1 - e^{-\lambda(f) c_f}) \prod_{i=1}^{k} (1 - e^{-\lambda_{0,i} c_{0,i}}) \tag{2.17}$$

2.9.5 Application Reliability

The expression for reliability, for an application with voltage scaling, is the same as the one presented section 2.7 for applications without replication. However the expression is repeated here for completeness:

$$R_{\mathcal{A}} = \prod_{P_i \in \mathcal{A}} R_{P_i} \tag{2.18}$$

2.9.6 Reliability Example

To show the use of the presented reliability expressions, the reliability for a process at three different voltage levels is calculated here. The process used, is the one previously shown in figure 2.2. In the examples a failure rate of $1.0 \cdot 10^{-6}$ is used.

Firstly, we evaluate the reliability of the process run at full speed with no fault tolerance. This is calculated using equation (2.2):

$$R_{Single} = e^{-\lambda c} = e^{-2.0 \cdot 10^{-6}} = 0.9999980 = 9\ nines\ and\ 8 \tag{2.19}$$

Now we choose to replicate the process to handle one fault ($k = 1$), such that it is run simultaneously on two processing elements. Both replicas are run at 66% voltage. The reliability is calculated using equation (2.16):

$$R_{Rep} = 1 - \prod_{i=1}^{k+1} (1 - e^{-\lambda(f)_i c_i}) = 1 - (1 - e^{-\lambda(66) c_i})^2 \tag{2.20}$$

Using equation (2.10) and assuming that $d = 2$ we find that:

$$\lambda(0.66) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} = \lambda_0 10^{\frac{2(1-0.66)}{1-0.34}} = 10\lambda_0 \tag{2.21}$$

and using equation (2.1) the duration of the scaled process is calculated to:

$$c_{66,i} = \frac{c_i}{0.66} = \frac{2 \cdot 3}{2} = 3 \tag{2.22}$$

hence:

$$\begin{aligned} R_{Rep} &= 1 - (1 - e^{-\lambda(66) c_i})^2 = 1 - (1 - e^{-30.0 \cdot 10^{-6}})^2 \\ &= 0.9999999991 = 9\ nines\ and\ 1 \end{aligned} \tag{2.23}$$

Now we choose to achieve the same level of fault tolerance by using re-execution, and running the root execution at a mere 34% voltage. Using equation (2.14) and the expression for the scaled failure rate we find that the reliability of this is:

$$\begin{aligned} R_{Reex} &= 1 - (1 - e^{-\lambda_0 c_0})(1 - e^{-\lambda(0.34) c_s}) \\ &= 1 - (1 - e^{-2.0 \cdot 10^{-6}})(1 - e^{-600.0 \cdot 10^{-6}}) \\ &= 0.9999999988 = 9\ nines\ and\ 0 \end{aligned} \tag{2.24}$$

Using the above approach the reliability of any single process can be calculated, and using equation (2.18) these can be combined to produce the reliability for an entire application.

2.10 Software Model

The fault-tolerance implementation for a system is managed by the on-line scheduler. This section presents the software model for the processors for different fault tolerance techniques.

In this thesis three different scheduler implementations are used for scheduling with fault tolerance. These approaches were introduced in [9], as an extension to the transparent recovery technique used in [12]. In this section the three fault tolerant scheduler implementations are presented.

Each processing element of an architecture has an online scheduler. In accordance with a pre-calculated *static*-schedule (or set of schedules) the scheduler will run processes. The scheduler detects whether faults occur, and similarly is responsible for executing the recovery processes, also in accordance with the static schedule. This is common for all three schedulers, but the way the static schedules are rendered and are handled, differs greatly between the three.

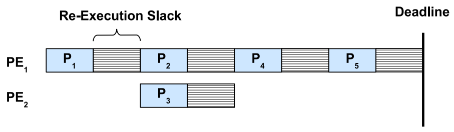
2.10.1 Fully Transparent Scheduler

This is the simplest, straightforward, implementation. After each process P_i, a recovery slack of length $kc_{P_{i,0}}$ is scheduled. That means that enough time is scheduled to run k re-executions and thereby handle k-faults.

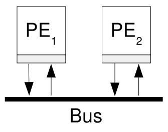
Thus, processes are scheduled with a free time slot after it, of a size which allows this process to re-execute on failing. This allows for using a single static schedule table, and the online scheduler will only have to detect whether a process fails and simply re-run it if it does. The slack in the schedule table allows for this to be done without any other process having to be delayed.

This scheduler implementation is fully transparent to fault occurrences, i.e. no information about faults has to known be the schedulers to take decisions.

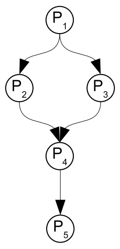
Figure 2.8 shows the process graph from section 2.3 scheduled with full transparency. Each process has recovery slack scheduled after it, and no process may



(a) Schedule



(b) Architecture



(c) Process graph

Process	PE_1	PE_2
P_1	2	2
P_2	2	2
P_3	2	2
P_4	2	2
P_5	2	2

(d) Durations

Process	Start Time
P_1	0
P_2	4
P_3	4
P_4	8
P_5	12

(e) Schedule table

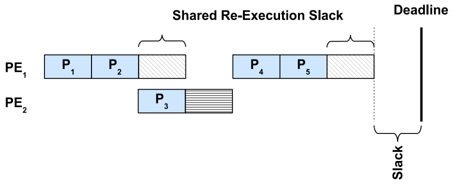
Figure 2.8: Fully transparent schedule for $k = 1$. The schedule shown is the fastest possible for this system configuration.

start until it is guaranteed that all processes before has had time to re-execute.

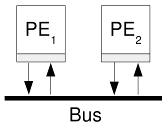
2.10.2 Slack Sharing Scheduler

The slack sharing scheduler sacrifices some of the transparency in order to achieve better performance. As the fault model dictates that no more than k faults will occur within a single execution, it is not necessary to handle more than this. The slack sharing scheduler exploits this information, by allowing processes on the same processing element to share re-execution slack. Figure 2.9 shows the same system scheduled using slack sharing. In the schedule we see that e.g. P_4 and P_5 share re-execution slack. As $k = 1$, only one of the two processes may experience a fault, and hence only a single recovery slack needs to be scheduled.

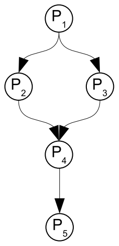
In this scheduler, fault information is shared on the local processor, but faults are still transparent between processors. In this way, process P_3 has to wait until time 4 to start, to ensure that process P_1 has had time to recover.



(a) Schedule



(b) Architecture



(c) Process graph

Process	PE_1	PE_2
P_1	2	2
P_2	2	2
P_3	2	2
P_4	2	2
P_5	2	2

(d) Durations

Process	Start Time
P_1	0
P_2	2
P_3	4
P_4	8
P_5	10

(e) Schedule table

Figure 2.9: Slack sharing schedule for $k = 1$. The schedule shown is the fastest possible for this system configuration.

2.10.3 Conditional Scheduler

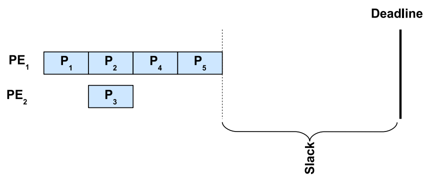
Fault tolerant scheduling using conditional scheduling has no transparency, i.e. all online schedulers share the information of faults. This allows the schedulers to respond very efficiently to faults, and hence produce schedules of high efficiency. In order to do this efficiently, a static schedule has to be created for each fault scenario. The possible schedules for the previous example, are shown in figure 2.10. These capture all possible fault scenarios. We see that the schedules are very efficient as only exactly k slacks are scheduled.

In order to capture all these different possible schedules an advanced conditional schedule table is needed (shown in figure 2.10(k)). The online schedulers will always start by executing the failure free schedule, marked *true*. If a fault is detected *all* online schedulers are notified and they will all switch to the corresponding recovery (contingency) schedule, marked F_{P_1} through F_{P_5}.

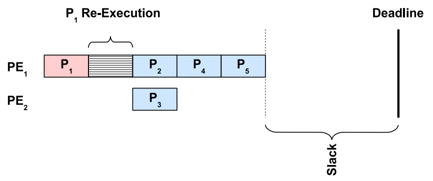
The conditional fault tolerant scheduling gives good control in terms of only scheduling the minimum amount of recovery slack. However this comes at the cost of the need of having more advanced online schedulers, more memory to store the larger schedule tables, and that fault information has to be shared between all processors, which increases bus utilisation. The broadcast of conditions on the bus is ignored in this thesis. However, we assume that the online schedulers can only make decisions based on the fault information they have at a given time. That is, the schedulers do not use information about faults that have not yet occurred.

From figure 2.10 it is seen that several of the possible schedules have the same deadline, and in fact are examples of worst case scenarios. Whenever conditional schedules are shown in the rest of this thesis, they will be one of such worst-case schedules.

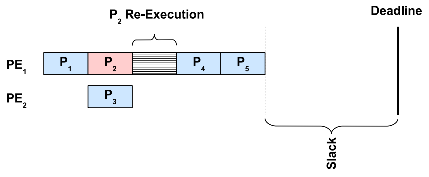
All processes are notified of all failures. In this way the schedule table for all nodes may/will change in the event of a process failure.



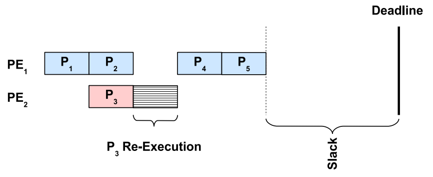
(a) No Fault



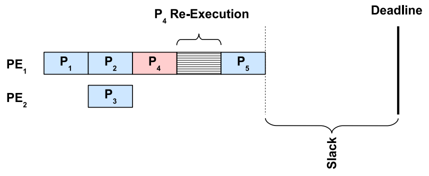
(b) P_1 Failing



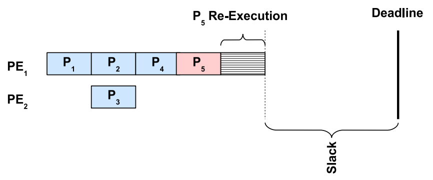
(c) P_2 Failing



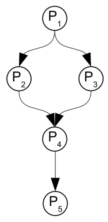
(d) P_3 Failing



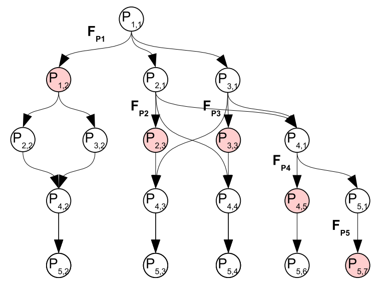
(e) P_4 Failing



(f) P_5 Failing



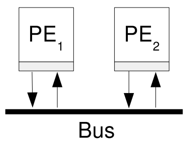
(g) Process graph



(h) Derived FT-CPG Taskgraph for $k = 1$

Process	PE_1	PE_2
P_1	2	2
P_2	2	2
P_3	2	2
P_4	2	2
P_5	2	2

(i) Durations Table



(j) Architecture

	Condition					
Process	**true**	F_{P_1}	F_{P_2}	F_{P_3}	F_{P_4}	F_{P_5}
P_1	0	0, 2				
P_2	2	4	2, 4			
P_3	2	4		2, 4		
P_4	4	6	6	6	4, 6	
P_5	6	8	8	8	8	6, 8

(k) Conditional Schedule Table

Figure 2.10: Illustration of the different possible schedules captured by a Fault Tolerant Conditional Process Graph. Figure 2.10(a) through 2.10(f) shows the Gantt charts for the possible fault scenarios, and table 2.10(k) shows the corresponding conditional schedule table (Blank entries are to be executed at the time specified in the **true** column).

Chapter 3

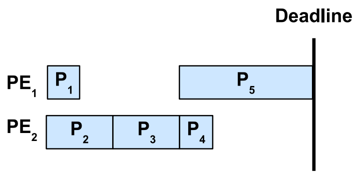
Problem Formulation

In this section motivational examples for the problems addressed in this thesis are presented. In the last section I give an exact problem formulation.

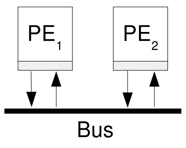
3.1 Complete Search vs. List Scheduling

Previous work in doing fault tolerant scheduling, using conditional process graphs, has been presented in [9]. In this work scheduling is done using the well known *list scheduling* algorithm. The list scheduling algorithm is a fast heuristic search implementation which offers good solutions to scheduling problems. However, as with any heuristic, the algorithm is not guaranteed to produce optimal results, and may hence produce a good schedule but not the globally optimal. In order to produce the optimal solution, it is necessary to explore *all* possible solutions. As scheduling problems are inherently NP-complete [29], this exploration is very costly in terms of time.

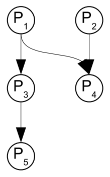
Using a complete search implementation to find optimal schedules may produce solutions of significantly better quality than list scheduling. As the embedded systems considered in the thesis all use static schedules, the extra expense, to do complete search, is a one-time cost, and may prove it self well worth it. Consider the process graph shown in figure 3.1(d) with the execution times shown in figure



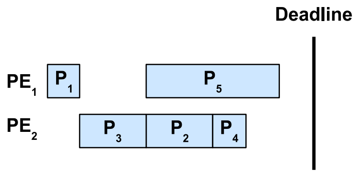
(a) List scheduling.



(c) Architecture



(d) Process graph



(b) Optimal scheduling.

Process	PE_1	PE_2
P_1	2	-
P_2	-	4
P_3	-	4
P_4	-	2
P_5	8	-

(e) Durations

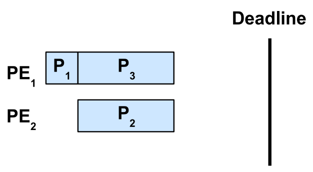
Figure 3.1: Performance of *list scheduling* versus optimal scheduling.

3.1(e). A "-" in the durations table denotes that a process cannot be mapped on that processing element. Given this input, a list scheduling algorithm will produce the solution shown in figure 3.1(a). List scheduling always selects a process if it is ready for execution, i.e. all its predecessors have already been scheduled. Hence list scheduling will schedule P_2 to start at time 0, as it has no predecessors. However this proves to yield a suboptimal solution. The optimal schedule is shown in figure 3.1(b). This shows that introducing some initial slack, also called idle time, on processor PE_2 is actually beneficial.

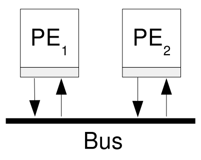
3.2 Policy Assignment

Determining whether a process is to be scheduled using re-execution or replication is called policy assignment. The problem of doing good policy assignment is critical in the optimisation process. The following example illustrates this importance.

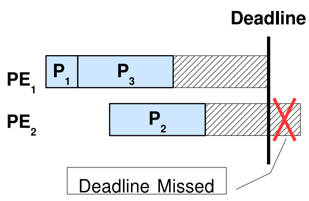
For a given architecture (figure 3.2(d)) an application (figures 3.2(e) and 3.2(f)) with a pre-defined mapping is to be scheduled under a reliability goal R_g. The fastest schedule for the application without fault tolerance is shown in figure 3.2(a). This schedule finishes well within the applications deadline, but does not meet the reliability goal. The designer hence wants to introduce redundancy to meet the reliability goal, and thus will introduce re-executions of all processes utilising slack sharing. The result of naively doing this is shown in figure 3.2(b)



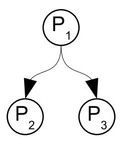
(a) No fault tolerance



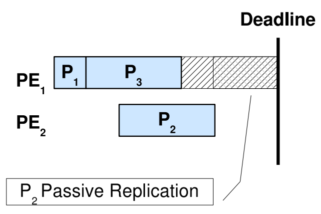
(d) Architecture



(b) Only re-execution



(e) Process graph



(c) Re-execution and passive replication

Process	PE_1	PE_2
P_1	2	3
P_2	4	6
P_3	6	9

(f) Durations Table

Figure 3.2: Illustration of the importance of considering redundancy assignment while scheduling. The application shown in figure 3.2(e) with the durations shown in 3.2(f) is to be scheduled on the architecture shown in figure 3.2(d). Figure 3.2(a) shows the fastest schedule with no fault tolerance. This however does not meet the reliability goal. Figure 3.2(b) shows the same system scheduled with redundancy to handle one fault, however the deadline is no longer met. To remedy this the re-execution of P_2 is passively replicated on the faster PE_1, this is shown in figure 3.2(c).

in which the reliability goal is met, but the deadline is now missed. Figure 3.2(c) shows the same application with an optimal use of both re-execution and replication. The redundancy of P_2 is now moved to the faster PE_1, and the application now meets its reliability goal and its deadline.

This example shows that it is important to consider the policy assignment when doing scheduling as it may drastically impact on the quality of produced schedules.

3.3 Power Consumption for Fault Tolerant Schedulers

The choice of fault tolerance scheduler implementation has great impact on the length of the produced schedule. This means that the choice of scheduler affects the amount of slack for use with voltage scaling to obtain energy savings.

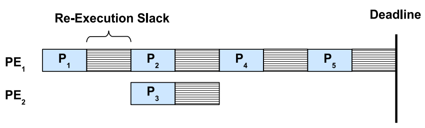
Figure 3.3 shows an application (figure 3.3(g)) that is to be scheduled on an architecture (figure 3.3(h)). The application is considered scheduled with each of the three schedulers: fully transparent, slack sharing, and conditional. The fastest possible solutions for each of these schedulers are shown in the left most column of Gantt charts. It is easily seen that the amount of available slack greatly increases with the use of more sophisticated schedulers. Slack sharing yields a slack of 2, where as conditional yields a slack of 6.

In this example I only consider the energy consumed by the root schedule. This is an approximation of the actual energy consumption, but describes the best case consumption. As it is only the root schedule that is subject to voltage scaling, it is only this energy that will give rise to energy savings. Hence, this is a reasonable simplification that makes the evaluation of achievable energy savings much easier. As the three schedules in figures 3.3(a), 3.3(c), and 3.3(e) all describe the same application, and all processes are run at the same voltage level, they naturally have the same energy consumption $E_{\mathcal{A},0}$.

We wish to exploit the available slack to do voltage scaling, and minimise the energy consumption. As there was no available slack when using fully transparent scheduling, no voltage scaling can be done and energy consumption is unchanged.

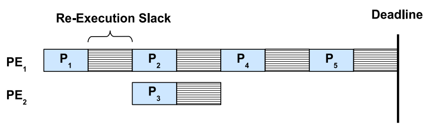
The slack sharing scheduler yielded a slack of 2, and hence energy optimisation is possible. The energy optimal schedule is shown in figure 3.3(d). From the Gantt chart it is seen that the optimal voltage scaling is to run processes P_1, P_3, P_4,

Fastest Schedule **Minimum Energy Schedule**



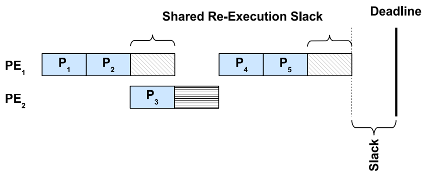
$Energy = E_{\mathcal{A},0}$

(a) Fully transparent



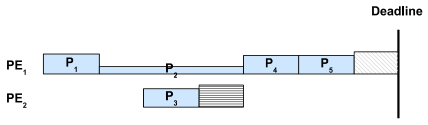
$Energy = E_{\mathcal{A},0}$

(b) Fully transparent



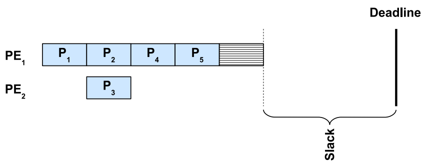
$Energy = E_{\mathcal{A},0}$

(c) Slack Sharing



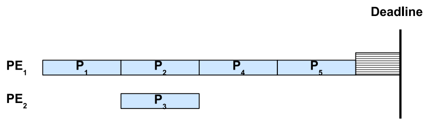
$Energy = 53\% E_{\mathcal{A},0}$

(d) Slack Sharing



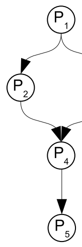
$Energy = E_{\mathcal{A},0}$

(e) Conditional

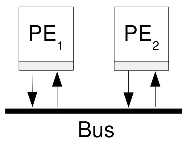


$Energy = 33\% E_{\mathcal{A},0}$

(f) Conditional



(g) Process graph



(h) Architecture

Process	PE_1	PE_2
P_1	2	2
P_2	2	2
P_3	2	2
P_4	2	2
P_5	2	2

(i) Durations

Figure 3.3: Illustration of the obtainable energy savings for different fault tolerance schemes. The first column shows the fastest schedules for each fault tolerance scheduling. The second column shows energy optimal schedules for the corresponding fault tolerance. The figures illustrate how more energy can be saved by using more advanced fault tolerance techniques which generate more slack. All the shown schedules can tolerate one transient fault, and have the same deadline, but the consumed energy to achieve this varies greatly with the different schedules.

and P_5 at the same frequency f_A, and P_2 at frequency f_{P_2}. The relations for the frequency levels is derived as the following two equations:

$$\begin{aligned} Deadline = 16 &= 3c_{f_A} + c_{f_{P_2}} + c_0 && (3.1) \\ Deadline = 16 &= 4c_{f_A} + 3c_0 && (3.2) \end{aligned}$$

Solving this we find the execution times of the scaled processes:

$$\begin{aligned} c_{f_A} &= \frac{16-3c_0}{4} = \frac{10}{4} && (3.3) \\ c_{f_{P_2}} &= 16 - 3c_{f_A} - c_0 = 16 - 3\frac{10}{4} - 2 = \frac{26}{4} && (3.4) \end{aligned}$$

Using equation (2.1) the frequencies they are run at can be found:

$$\begin{aligned} f_A &= \frac{c_0}{c_{f_A}} = \frac{2 \cdot 4}{10} = 80\% && (3.5) \\ f_{P_2} &= \frac{c_0}{c_{f_{P_2}}} = \frac{2 \cdot 4}{26} = 31\% && (3.6) \end{aligned}$$

With these values the energy consumption for processes at the two frequencies can be calculated. To do so equation (2.8) is used. As the deadline is fixed for the application the frequency independent power would contribute with the same amount for any schedule, and is hence disregarded of. The power exponent is assumed to be $m = 3$ which is a reasonable value [30]:

$$\begin{aligned} E_{P_2} &= C_{eff} f_{P_2}^m c_{P_2} = C_{eff}\left(\frac{8}{26}\right)^3 \frac{26}{4} = 0.189C_{eff} = 9.4\%E_0 && (3.7) \\ E_{P_1} &= C_{eff} f_A^m c_A = C_{eff}\left(\frac{8}{10}\right)^3 \frac{10}{4} = 1.28C_{eff} = 64\%E_0 && (3.8) \end{aligned}$$

Where E_0 is the energy consumed by a process run at full speed. Summing the energy contributions we find the total energy consumed for the schedule, using equation (2.9):

$$E_{\mathcal{A},slack} = 4E_{P_1} + E_{P_2} = (4 * 1.28 + 0.189)C_{eff} = 5.31C_{eff} = 53\%E_{\mathcal{A},0} \quad (3.9)$$

Using the slack sharing scheduler we can achieve a system with the same level of fault tolerance as for the transparent schedule, but which only consumes 53% of the energy.

The conditional scheduler yielded a slack of 6 and hence has even greater potential for voltage scaling. The energy optimal conditional schedule is shown in figure 3.3(f). From the schedule we see that:

$$\begin{aligned} & Deadline && = 16 = 4c_{f_A} + c_0 && (3.10) \\ \Downarrow \\ & c_{f_A} && = \frac{16 - c_0}{4} = \frac{14}{4} && (3.11) \end{aligned}$$

Which leads to:

$$f_A = \frac{c_0}{c_{f_A}} = \frac{2 \cdot 4}{14} = 57\% \quad (3.12)$$

From this we find the energy consumption for the conditional scheduler to be:

$$E_{\mathcal{A},cond} = 5C_{eff} f_A^m c_{f_A} = 5C_{eff} \left(\frac{8}{14}\right)^3 \frac{14}{4} = 3.27C_{eff} = 33\%E_{\mathcal{A},0} \quad (3.13)$$

The conditional scheduler gives the same level of redundancy, but consumes only 33 % of the energy of the transparent scheduler.

We see that, using more advanced fault recovery schedules, the energy consumption of an application can be reduced dramatically, while the level fault tolerance is maintained.

3.4 Reliability and Scheduling

As shown in previous sections 3.2 and 3.3 scheduling has great impact on the obtainable slack, and thereby the amount of voltage scaling that can be performed. In this example the effect of voltage scaling on system reliability is investigated.

Figure 3.4 shows an application to be scheduled such that energy consumption is minimised, while a required reliability goal of 9 nines is met.

Using slack sharing scheduling the fastest schedule is shown in figure 3.4(a). The energy is $E_{\mathcal{A},0}$, and the deadline is met, however the energy consumption is not minimised. The reliability for the schedule is calculated using equations (2.14) and (2.16) in similar manner to the example in section 2.7, using the constants $d = 2$, $\lambda_0 = 10^{-6}$ and the frequencies listed in figure 3.4(e).

Optimising the application for minimum energy consumption, with the deadline as a hard constraint, results in the schedule shown in figure 3.4(b). Due to the probability of faults being dependent on the frequency, the reliability of the system is lowered. The probability of error is increased by:

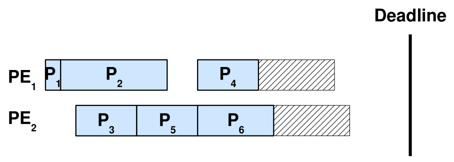
$$\Delta\rho = \frac{1 - R_{min\_energy}}{1 - R_{fastets}} \simeq 13 \tag{3.14}$$

The consumed energy is reduced to merely 56% of the fastest schedule, but the reliability goal is missed.

To ensure meeting the reliability goal, this is imposed as a hard constraint along with the deadline. The optimal schedule under these constraints is shown in figure 3.4(c). Now all constraints are met and, under these, the energy is minimised. For this schedule the energy is reduced to 74% of the energy for the fastest schedule. In order to produce a minimal energy schedule under the reliability goal, the processes on PE_2 are forced to swap places.

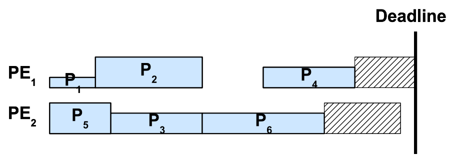
This example shows that reliability has to be considered at the same time as doing scheduling and voltage scaling in order to produce optimal schedules. If this is not done, the designed system may become very unreliable. Further, optimal schedules under reliability constraints will need to sacrifice some energy savings in order to be reliable.

However, my optimisation algorithms are able to produce schedules with constrained reliability, which yield energy savings comparable to schedules with



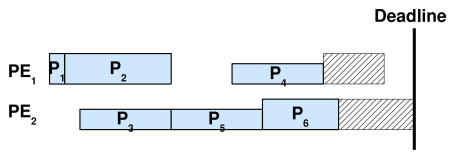
Finish Time	20 ms
Energy	$E_{\mathcal{A},0}$
Reliability	10 nines

(a) Fastest Possible Schedule



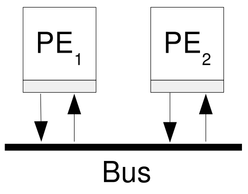
Finish Time	24 ms
Energy	$55.7\% E_{\mathcal{A},0}$
Reliability	8 nines and 4

(b) Minimum Energy



Finish Time	24 ms
Energy	$73.7\% E_{\mathcal{A},0}$
Reliability	9 nines and 2

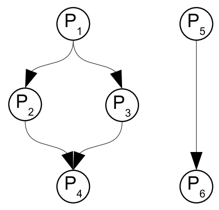
(c) Constrained Reliability, 9 nines



(d) Architecture

	Voltage Level		
PE_1	100 %	67 %	34 %
PE_2	100 %	67 %	34 %

(e) Voltage Levels



(f) Process graph

Process	PE_1	PE_2
P_1	1	1
P_2	7	7
P_3	4	4
P_4	4	4
P_5	4	4
P_6	5	5

(g) Durations Table

Figure 3.4: Example of the necessity of considering reliability when doing scheduling and energy optimisations. The application is shown in figure 3.4(f) with the corresponding process durations in figure 3.4(g). The application is to be mapped onto the architecture shown in figure 3.4(d) with corresponding voltage levels in figure 3.4(e) using slack sharing fault tolerance scheduling. In figure 3.4(a) the schedule has been optimised for speed alone. Figure 3.4(b) shows the same system optimised for minimal energy consumption. Note that P_3 and P_5 have swapped places on PE_2 to allow for better voltage scaling. Finally figure 3.4(c) shows the system optimised for minimal energy consumption under the reliability goal of 9 nines. Again the processes swap places to allow for the best scaling under this constraint.

unconstrained reliability.

3.5 Problem Formulation

Considering an application $\mathcal{A}$ with the process graph $\mathcal{G}$, and a distributed architecture consisting of a number of processing elements connected by a single bus, we determine a reliability goal corresponding to fault-tolerance for k transient faults. For this system we wish to perform the following design tasks. Create a schedule, i.e. determine the start time for each process. Do mapping, that is the allocation of each process onto processing elements. Do voltage scaling to minimise the energy consumption. Apply a fault-tolerance policy, either re-execution or passive replication, to each process such that the applications is tolerant to k transient faults. All of these tasks have be considered simultaneously to produce a design in which the application is schedulable, the energy is minimised, and the reliability goal is met.

Chapter 4

Energy-Optimisation under Reliability and Timing Constraints

In this chapter I present the optimisation algorithms for energy minimisation under reliability and timing constraints. Section 4.1 and 4.2 introduce constraint logic programming and ECL^{i}PS^e respectively. Section 4.3 presents the logic constraints that correspond to general embedded systems design tasks. The constraints specific to fault tolerance are presented in section 4.4. The objective function of the optimisation is described in section 4.5. The search strategy used in the optimisation is presented in section 4.6.

4.1 Constraint Logic Programming

Linear programming (*LP*) has been a popular tool for modelling and doing optimisations for many years, especially in *operations research*. *LP* models are composed by a set of algebraic equations which describe the system. To find a solution, general purpose solvers are used to search the design space. This has the great advantage that all models will be able to use the same solver, and hence improvements to this solver can be shared by all users of the *LP* system.

This has given rise to very sophisticated solvers with very good performance.

However, algebra is a very limited tool for modelling. A much more powerful mathematical tool is logics. Logic programming is also a well established approach, based on languages such as PROLOG. Logic programming is conceptually identical to *LP* in that a set of rules describe a solution, and a solver searches to find this solution. However for many applications outside artificial intelligence, logic programming has rather poor performance.

Recent years have seen the advent of a hybrid of the two paradigms, the constraint logic programming (*CLP*). The added constraints allow for specifying algebraic constraints on the systems defined by the use of logics. This has given rise to *CLP* systems with the modelling capabilities of logics, as well as the performance of *LP*. *CLP* has especially proven to yield good performance in solving NP-hard problems.

Programming in *CLP* is based on logic constraints. A system is described by a set of 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.

4.2 The ECL^{i}PS^e-*CLP* System

ECL^{i}PS^e is a PROLOG based *CLP* system. Its logic kernel, and all components for it are programmed in PROLOG. The actual programming language is however not standard PROLOG, but offers constructs specific to *CLP*.

ECL^{i}PS^e was originally developed by the European Computer-Industry Research Centre (ECRC) in Munich and later by IC-Parc at Imperial College of London, but has been open-sourced in the summer of 2006 and is now publicly available as a community project [5] supported by Cisco Systems.

The ECL^{i}PS^e system includes a wide variety of libraries and extensions, as well as a simple *backtracking* solver. The language offers primitives to allow for easy construction of custom solvers, both for complete search, as well as heuristics.

Further details about ECL^{i}PS^e are available in [1, 5].

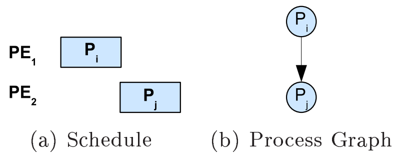


Figure 4.1: Illustration of the precedence constraint. Process P_j in the process graph cannot start untill it has received data from its predecessor P_i.

4.3 Constraints for Embedded Systems

In this section I present a series of logic constraints, which have been used in the literature to model embedded systems design tasks. These model general things like processor and data behaviour, and are as such applicable to all types of systems. The constraints specific to fault tolerance are presented in the next section.

4.3.1 Precedence Constraints

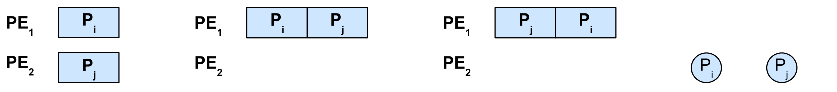
The sequence of processes in an application $\mathcal{A}$ is determined by their inter-communications $\mathcal{E}$ (see section 2.3 for definition). No process can be executed before all the processes, from which it depends on communications from, have been executed. Recalling that an edge e_{ij} denotes a communication from process P_i to P_j the precedence constraint can be formalised as:

$$Start(P_j) >= \forall_{e_{ij}} Start(P_i) + Duration(P_i) \tag{4.1}$$

which must hold for all processes $P_j \in \mathcal{A}$. An example of the constraint is shown in figure 4.1. The example shows how the data dependency e_{ij} in the process graph forces P_j to start after P_i has finished.

4.3.2 Resource Constraint

The resource constraint enforces the constraint that a processor can only execute a single process at a time. Two processes may either be executing on different processors, or execute such that their executions do not overlap in time. This is formally expressed by:



(a) Schedule A (b) Schedule B (c) Schedule C (d) Process graph

Figure 4.2: Illustration of resource constraint. Two processes cannot occupy the same processing element at the same time. The three Gantt charts illustrate valid schedules.

$$
\begin{aligned}
&Mapping(P_i) \neq Mapping(P_j) \\
&\vee Start(P_i) \neq Start(P_j) + Duration(P_j) \\
&\vee Start(P_j) \neq Start(P_i) + Duration(P_i)
\end{aligned}
\tag{4.2}
$$

which must hold for all process pairs P_i and P_j where $i \neq j$. Three schedules, that adhere to this constraint, are shown in figure 4.2. Two processes with no dependencies are to be scheduled. They can either be mapped on different processors (figure 4.2(a)), as expressed by the first clause of the constraint, they may execute non-overlappingly on the same processor (figure 4.2(b) and 4.2(c)).

4.3.3 Timing, Reliability and Energy Constraints

Further, all variables concerning time, can be constrained to be within the deadline. For the start times of processes this can be formally written as:

$$Start(P_i) + Duration(P_i) \leq Deadline \tag{4.3}$$

which must hold for all processes $P_i \in \mathcal{A}$. In fact, this constraint must only hold for the end process(es) of an application. Specifying it for *all* processes, allows the underlying *CLP* engine to restrict the possible values for the timing variables. This, in turn, makes it easier for the solver to prove optimality, and makes searching for solutions faster.

The constraints for voltage scaling, reliability, and energy are direct implementations of the equations, presented in sections 2.3, 2.7 and 2.8 respectively, and are not repeated here. The use of these equations are shown in examples in section 2.9.6 and 3.3.

4.4 Constraints for Fault Tolerance

In this section the constraints specific to fault tolerance are presented. The presented constraints form an addition to the general constraints presented in the previous section, and as such are an incremental addition to add support for fault tolerance to the already presented model. The constraints are presented individually per fault tolerance technique. Examples of schedules with each of the presented techniques can be found in figure 3.3.

4.4.1 Fully Transparent Scheduler

In fully transparent scheduling, recovery slack is scheduled after each process. This is modelled by setting the length of a process to the length of the root execution, *plus* the length of k recovery executions:

$$Start(P_j) >= \forall_{e_{ij}} Start(P_i) + Duration(P_i)(1+k) \tag{4.4}$$

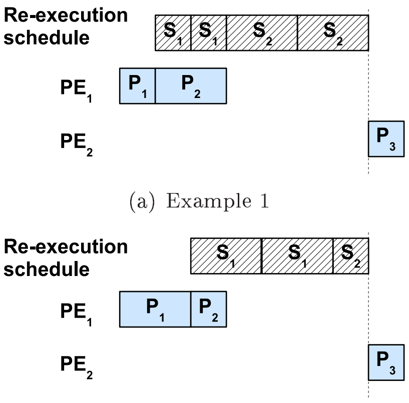
Which must hold for all processes $P_j \in \mathcal{A}$. This is an adaption of the precedence constraint from section 4.3.1. If voltage scaling is applied, this will effect only the length of the root execution, as all re-executions will still be executed at full speed. The expression is then:

$$Start(P_j) >= \forall_{e_{ij}} Start(P_i) + Duration_f(P_i) + Duration(P_i)k \tag{4.5}$$

Which again must hold for all processes $P_j \in \mathcal{A}$.

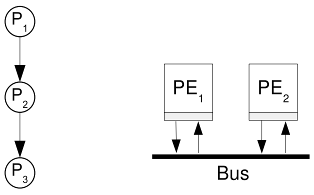
4.4.2 Slack-Sharing Scheduler

When scheduling with fault tolerance using the slack sharing technique, processes with dependencies on the same processor and on other processors need to be treated differently. The constraints for each of these cases is hence treated separately in the following.



(a) Example 1

(b) Example 2



(c) Process Graph (d) Architecture

	Example 1		Example 2	
Process	PE_1	PE_2	PE_1	PE_2
P_1	2	-	4	-
P_2	4	-	2	-
P_3	-	2	-	2

(e) Durations

Figure 4.3: Illustration of the availability of data in a slack sharing schedule, for processes scheduled on different processing elements. The Gantt charts show the critical re-execution schedule.

Processes on the Same Processing Element

Processes executed on the same processor share recovery slack. This slack will always be scheduled after the root processes. Hence the root processes can be scheduled without any further constraints. Thus the constraint for processes on the same processing element is simply:

$$Mapping(P_i) = Mapping(P_j) \tag{4.6}$$

Processes on Different Processing Elements

Things are more complex if the two processes are mapped on different processors. As just described, processes on the same processor share recovery slack. Processes on different processors however cannot be started until recovery of their precedents is guaranteed.

The situation where to processes on different processors have to communicate, can be split into two special cases. These are illustrated in figure 4.3. The constraints for the two cases are presented below individually.

data is available to be transmitted to another process can be described by two special cases, both illustrated in figure 4.3, and the constraints are presented below.

Example 1: We consider the dependency between process P_2 and P_3. In figure 4.3(c), P_2 is scheduled after a shorter process. 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:

$$Start(P_3) >= Start(P_2) + Duration_f(P_2) + Duration(P_2)k \tag{4.7}$$

Example 2: In figure 4.3(d), P_2 is scheduled after a longer process. 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 process scheduled before this. The start time of P_3 is constrained by:

$$Start(P_3) >= Start(P_1) + Duration_f(P_1) + Duration(P_1)k + Duration(P_2) \tag{4.8}$$

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 generalise the shown constraints, in a way that can be used in an *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 figure 4.3, the created recovery schedule in fact is identical to the recovery schedule shown in each Gantt chart.

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 are imposed as presented in section 4.3.1 and 4.3.2. The finishing times of the processes in the recovery schedule is described by:

$$\begin{aligned} Finish(S_i) \geq Start(P_i) + Duration_f(P_i) + Duration(P_i)k \\ \wedge Finish(S_i) \geq Start(S_i) + Duration(P_i) \end{aligned} \tag{4.9}$$

As seen in the previous example (especially figure 4.3(d)), the duration of the recovery process, is dependent on its predecessors. Hence the above constraint, cannot be written as elegantly in terms of the duration, and is hence kept in this form.

Note that the first part of the expression, up to the $\wedge$ operator, captures the recovery schedule in example 1. Similarly the rest describes example 2.

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

$$Start(P_j) >= Finish(S_i) \tag{4.10}$$

General Expression

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:

$$\begin{aligned} & Mapping(P_i) = Mapping(P_j) \wedge Start(P_j) \geq Start(P_i) + Duration_f(P_i) \\ \vee \quad & Start(P_j) \geq Finish(S_i) \end{aligned} \tag{4.11}$$

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

4.4.3 Conditional Scheduler

The conditional scheduler implementation is based on the *FT-CPG*s presented in section 2.6.

The use of constraint logic programming for scheduling conditional process graphs is described in [15]. The constraints presented in this section is an extension of that work to allow for scheduling *FT-CPG*s.

The conditional edges in the *FT-CPG* form mutually exclusive paths through the graph. As a consequence two process, which depend on mutually exclusive conditions, will never be executed in the same run of an application. As an example, consider $P_{5,2}$ and $P_{5,3}$ in figure 2.6(b). $P_{5,2}$ depends on the failure of $P_{1,1}$ and $P_{5,3}$ depends on $P_{1,1}$ *not* failing (and $P_{2,1}$ failing). Consequently these two processes will never be active in the same execution. Because of this, processes which are part of mutually exclusive paths can be scheduled to the same resource at the same time. This addition to the resource constraint from equation (4.2) is written:

$$\begin{aligned}
&MutuallyExclusive(P_i, P_j)\\
&\vee Mapping(P_i) \neq Mapping(P_j)\\
&\vee Start(P_i) >= Start(P_j) + Duration(P_j)\\
&\vee Start(P_j) >= Start(P_i) + Duration(P_i)
\end{aligned} \tag{4.12}$$

which must hold for all process pairs P_i and P_j where $i \neq j$. The function $MutallyExclusive$ determines whether the two processes are on two disjunctive paths (as described in the above paragraph). This function is computationally heavy, as it involves recursively searching through the lists of conditions for each process. These condition lists are created as part of deriving the *FT-CPG*. Therefore, the lists are available to optimisation tool when it loads the *FT-CPG*. As the conditions for processes are independent of the scheduling, the recursive search to determine mutual exclusiveness of processes can be done as a one time effort as part of the setup of the internal model. In the actual implementation the function to determine mutual exclusiveness is run first. If the two processes are mutually exclusive, they do not constrain each other, and nothing further is done. If they are part of the path, the constraints presented above are invoked. The logic expression shown above captures this behaviour concisely.

In [15], Kuchinski does conditional scheduling by using a graphical method to draw processes which depend on different conditions with different width. In his work, he only operates with a single condition, for which his approach works well, and is very intuitive, due to its visual resemblance to Gantt charts. However, in the application of fault tolerance, with the inherently large number of conditions, the graphical approach would become impractical (processes would become impractically "thick" in the graph). Further to use this graphical method Kuchinksi exploits a built-in predicate in the *CHIP* constraint logic programming system. This predicate is not available in ECL^{i}PS^e, and hence the graphical solution is not an option in this implementation. Most importantly, his implementation will evaluate mutual exclusiveness as part of model while searching for solutions. For large numbers of conditions, this will become very time consuming. The solution presented in this thesis is more efficient, as the costly comparison of conditions is only done a single time, as part of loading the model. This gives less logic constraints to evaluate at search time, and hence significantly faster search performance of the scheduler.

4.5 Objective Function

The tool uses reliability and deadline as hard constraints. The *CLP* solver uses this to constrain the design space, such that for any found solution, these two constraints will always be satisfied.

The optimisation can hence focus on the consumed energy alone. The equations used to express the energy are the ones presented in section 2.8. As the goal of the tool is to achieve energy savings from using power management techniques, we are only interested in getting a measure for this saving. The optimisation process only applies voltage scaling to the root schedule, and therefore only this will contribute to energy savings. Hence, the tool optimises the energy consumption of the root schedule only. This is the same approach as used in the example in section 3.3. This approach makes the evaluation of the energy simpler, and hence faster, while still enabling the tool to precisely determine the energy savings.

4.6 Search Strategy

A *CLP* program is composed by a set of logic constraints. To find solutions for such a model, the solver will search through all possible values of all variables, to find combinations of values which satisfy all constraints.

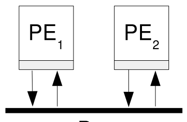
Consider a single process to be scheduled and mapped to run on an architecture with two processing elements, and be optimised for fastest execution. This example is illustrated in figure 4.4. The application has a deadline of 4, and the process has a duration of 5 on PE_1 and 1 on PE_2. It should be obvious that the process can only be mapped on PE_2 in order to meet its deadline, a point I shall return to shortly. To map and schedule this process the solver has two design tasks to decide: mapping and scheduling. Due to the constraint specified in equation (4.3), which states that the start time plus the duration of all processes will always be smaller than the deadline, the solver will limit the value space for the start times of the process, to the interval 0-3.

In the search tree in figure 4.4(d) the start times are considered first, and then, for each start time, the mappings are considered. This gives rise to the shown search tree, which has three internal-nodes, and six leaf-nodes (corresponding to

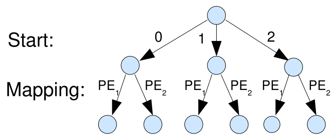
(a) Process graph

Process	PE_1	PE_2
P_1	1	5

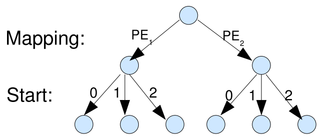
(b) Durations



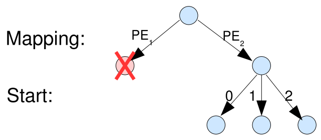
(c) Architecture



(d) Search tree 1



(e) Search tree 2



(f) Search tree with pruning

Figure 4.4: Illustration of search trees for a simple mapping and scheduling example. The application is to be mapped to the architecture. The schedule is to be optimised for speed, and finish within a deadline of 4.

the six permutations of the values for the two variables). An alternative search tree where mappings are considered first, and then start times, is shown in figure 4.4(e). This tree has only two internal nodes, yet naturally the same six leaf-nodes. Hence, this tree is preferable to the other, as it will end up at the same results, but visit less internal states in the process, and hence be faster. This shows that the order in which the design tasks are performed, has a big impact of the number of states that needs to be visited while searching for solutions.

Let us return to the fact that the process runs too slowly on PE_1. Using the ordering of design tasks from the optimal search tree from figure 4.4(e), and a solver that will always select the smallest value first, the solver will perform the search shown in figure 4.4(f). In this tree, the process is firstly mapped on PE_1. The solver evaluates the constraints, and finds that due the process' duration being longer than the deadline of the application, this mapping is not valid. It hence backtracks and maps the process on PE_2 instead. While doing this, the solver does *not* try any values for start times. These branches are cut of, or pruned. With the new mapping the solver will try the three different start times, and determine that 0 yields the fastest schedule. Using pruning, the solver only had to visit 6 nodes in the search tree, out of the total 9. This illustrates that the order in which the design tasks are performed, affects both the number of internal nodes in the search tree, and the solvers ability to do efficient pruning. These two things both have very significant performance impact.

For the proposed optimisation algorithm, it has been found that performing design tasks in the following order yields best search performance: mapping, voltage scaling, and scheduling.

In the previous example we used a solver which considered variable values, from the smallest value and up. This is not always the best strategy. Accelerating searches by changing the sequence in which values are considered, is called *value selection*, and is a search heuristic, which does not sacrifice optimality. The implemented solver in the presented work, uses the following value selection schemes for each variable: Mappings: random, Voltage levels: from the minimum, and Start times: from the minimum. This has been found to yield the best performance. Evaluating the voltage levels from the smallest first, will bias the search towards finding the schedules with the least energy consumption first.

4.6.1 Optimality vs. Fast Solutions

For larger applications each design task consists of assigning values to a large number of variables, e.g. all processes will have a mapping and a start time variable. Each of these variables will be assigned a value using the strategy presented above, but the sequence in which the variables are chosen to be assigned is also an important part of the solver implementation. Speeding up search, by changing the way in which variables are chosen for assignment is called *variable selection*, and is too a search heuristic.

ECL^{i}PS^e offers predicates to implement a number of different variable selection schemes. The *most constrained* scheme, will select the variable in the current set, e.g. mappings or start times, that has most constraints associated with it. The variable will be assigned a value, and then the second most constrained variable is selected. This is repeated until the set is empty. A similar approach is the *first fail* scheme, which tries to guess which variable will be the first to lead to an invalid solution. This is done in a simpler and faster way than the *most constrained* scheme. The aim of these two approaches, is to try to find invalid solutions first, in an attempt to do pruning of the search tree as soon as possible. Early pruning, effectively reduces the size of the search tree, and consequently makes it easier for the solver to prove optimality of a solution (which it can only do after having visited all valid solutions). However as a lot of the initial searches are intentionally directed down search paths that hold no solutions, these two approaches may take very long time to find the first solution, however this first solution will often be the optimal.

As an alternative, ECL^{i}PS^e offers the *anti first fail* selection scheme, which will try to find a path that produces solutions as fast as possible. This approach will produce a lot of solutions, that gradually get better. But these will generally start out being *very* bad, as the variable selection choose the paths through the search tree that represent the *easy* solutions. This selection scheme may produce good solutions, but will take much longer time to prove them optimal. This is because the variable selection scheme causes the search to visit a lot of solutions, which pruning could have shown to be suboptimal.

In the proposed implementation, the main interest is to find the optimal solutions, and I hence use the *first fail* variable selection approach for all design tasks. This has been found to yield a good trade-off between the speed of finding solutions, and the solvers ability to prove the optimality of solutions.

Chapter 5

Experimental Results

In this section I present the experiments performed, in order evaluate the proposed scheduling and optimisation approaches. The experiments have been conducted on two sets of test data, a set of synthetic applications presented in section 5.1, and a case study of an *MP3-decoder*, presented in section 5.2. The remaining sections of this chapter present the conducted experiments.

5.1 Synthetic Applications

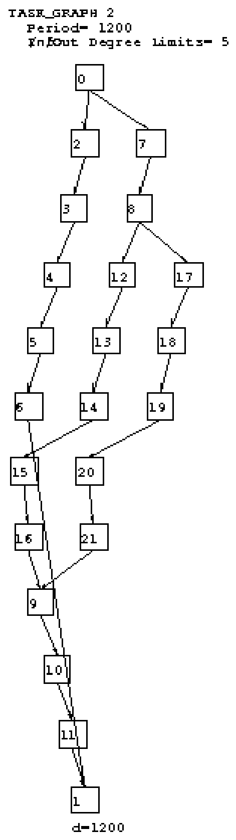
A large set of synthetic applications have been generated using the *task graphs for free* tool (*TGFF*) [4, 26]. This tool generates pseudo-random process graphs in a platform independent and general way, allowing researchers to experiment with their results, on similar input material. I have configured *TGFF* to generate *series parallel* graphs, which resemble graphs for real applications. In figure 5.1 a sample *TGFF* series parallel graph is shown, together with the input parameters used to create it.

The test set is composed of graphs with $N \in 10, 15, 20, 25, 30$ processes. For each graph size, I have generated a total of ten graphs. Half of the processes in the graphs have been randomly chosen to be made redundant. The remainder

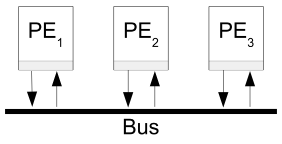
```
tg_cnt 5
task_cnt 15 1
gen_series_parallel true
period_laxity 1
period_mul 1, 1, 1

tg_write
eps_write
vcg_write
pe_write
```

(a) Input parameters



(b) Generated graph example



(c) Architecture

	Voltage Level		
PE_1	100 %	67 %	34 %
PE_2	100 %	67 %	34 %
PE_3	100 %	67 %	34 %

(d) Voltage levels

Figure 5.1: Parameters used for *TGFF* and an example of a corresponding generated process graph. Also the architecture used in the synthetic experiments is shown.

of the processes are considered non-critical, and are not made redundant.

Where not explicitly stated otherwise, the architecture shown in figure 5.1(c) is used. The architecture consists of three processing elements, connected by a single bus. Each processing element can be run at three voltage levels. The applications have been randomly mapped unto the architecture.

In the experiments, the fully transparent schedule has been used as reference. This schedule is the straightforward approach, that an experienced designer would determine, without using my tool. The deadline for the graphs in the experiments, has been set to the length of the optimal fully transparent schedule.

Similarly, the reliability goal is determined based on the reliability of the fastest fully transparent schedule. The reliability goal is defined as:

$$R_g = 1 - 10(1 - R_{transparent}) \tag{5.1}$$

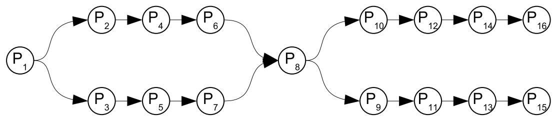
which means, the probability of faults may be no more than ten times greater than in the transparent schedule.

5.2 *MP3-decoder* Case Study

The experiments have also been conducted on a real application. This is an *MP3-decoder*, for which a process graph, as well as detailed timing information is available. This example has previously been used in [18] and [24].

The process graph for the *MP3-decoder* is shown in figure 5.2. The graph has two parallel executions, with identical durations, and two intersects. This is because the decoded *MP3*-stream is stereo, and the two channels are decoded independently. The durations shown in the figure are written as the number of cycles they need to complete. The deadline for the application is 25ms.

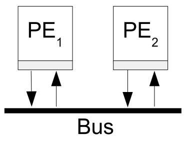
The *MP3-decoder* is executed on an architecture with two processing elements, shown in figure 5.2(c). The individual processors can be run at three voltage levels. The voltage levels have been set slightly higher than in the architecture for the random process graphs. This is due to processes P_{13} through P_{16} being relatively expensive in terms of execution time. With the lower voltage levels, these processes can not be voltage scaled within the deadline, and the slack can not be efficiently used for energy management.



(a) Process graphs

Description	**Process**	PE_1	PE_2
Preprocessing	P_1	1071	1071
Scale	P_2, P_3	476	476
Huffman Decoder	P_4, P_5	36781	36781
De-quantisation	P_6, P_7	14172	14172
III-Stereo	P_8	63914	63914
Reorder	P_9, P_{10}	2568	2568
Antialise	P_{11}, P_{12}	21305	21305
IDCT	P_{13}, P_{14}	144924	144924
Sub-Band Synthesis	P_{15}, P_{16}	266687	266687

(b) Durations



(c) Architecture

	Voltage Level		
PE_1	100 %	75 %	50 %
PE_2	100 %	75 %	50 %

(d) Voltage levels

Figure 5.2: Process graph for *MP3-decoder*. The descriptions are from [24]. The execution times are here listed as the amount of cycles they need to execute. The architecture the is also shown, along with the available voltage levels.

name	Constant	Value
Effective capacitance	C_{eff}	$1.11 \cdot 10^{-9}$ F
Power exponent	m	3
Frequency independent power	P_{ind}	0 mJ
Initial failure rate	λ_0	$1.0 \cdot 10^{-6}$ faults per second
Failure rate constant	d	2

Figure 5.3: Constants used in experiments. The failure rate is assuming a 100 megabit chip [32].

5.3 Optimisation Parameters

The algorithms have been evaluated considering two situations. In the first case, the application must tolerate one transient fault ($k = 1$), and in the second case, they tolerate two faults ($k = 2$).

All experiments have been conducted with a fault tolerance level of $k \in \{1, 2\}$. The constants used for the numerical calculations are shown in figure 5.3. All experiments are conducted with a hard deadline. The system is assumed to be online continuously, hence the frequency independent power, P_{ind}, can safely be set to naught, as it will only contribute with a constant to the energy expression from equation (2.9), with value $Deadline \cdot P_{ind}$.

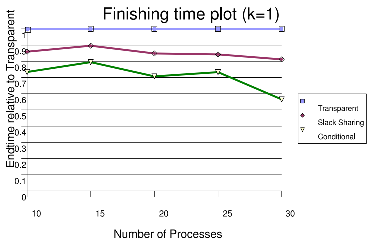
The constant values are taken from [30] and [19].

The *CLP* solver that searches for schedules, is set to have a timeout of 15 minutes. For some schedules optimality is proved within this deadline. Other searches may produce intermediate results, but not be able to prove optimality. Finally, some searches may not find any solutions within this deadline. How many searches fall into which category is listed for each experiment in the following.

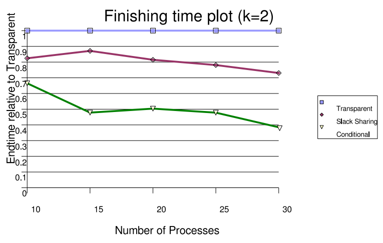
The experiments have been conducted using the ECL^{i}PS^e version `5.10_44`, running on 3.5 Ghz AMD 64-bit computers with 2 gigabytes ram.

5.4 Performance of Fault-Tolerant Schedulers

To compare the three implemented schedulers against each other, two experiment runs have been performed. In the first, the optimisation criteria is finish



(a) $k = 1$



(b) $k = 2$

	No. of Processes				
No. Faults	10	15	20	25	30
Fully transparent					
$k = 1$	10/0/0	9/1/0	7/3/0	5/3/2	2/5/3
$k = 2$	10/0/0	10/0/0	7/2/1	6/4/0	3/4/3
Slack sharing					
$k = 1$	10/0/0	9/1/0	7/3/0	5/3/2	2/4/4
$k = 2$	10/0/0	10/0/0	7/2/1	6/4/0	3/4/3
Conditional					
$k = 1$	4/6/0	2/8/0	0/10/0	0/8/0	0/7/3
$k = 2$	3/7/0	0/10/0	0/9/1	0/10/0	0/7/3

(c) Finishing status of searches

Figure 5.4: Comparison of the fastest possible schedules obtainable with the three schedulers. All values are relative to those of the fully transparent scheduler. Smaller values are better. The table shows the number of *optimal / intermediate / none* schedules for each graph size.

time, and their ability to produce fast schedules is compared. The results for this experiment is shown in the graphs in figure 5.4. The x-axis marks the size of the graphs, and the y-axis is the length the produced schedule, 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. The number of *optimal / intermediate / none* results for the searches are shown in the table in figure 5.4(c). Optimal results are those which have been found within the timeout. Since the search has stopped before the timeout, the found schedule is known to be optimal. Intermediate results are the best known schedule, when the search reached timeout. The searches marked as none, did not find any valid solutions within the timeout.

From the graphs we see that for systems tolerating one transient fault, the slack sharing approach produces results that are consistently 10-15% shorter than those for the fully transparent scheduler. The conditional approach is 20-30%

	Fault Tolerance Technique		
$k = 1$	**Fully Trans.**	**Slack Shr.**	**Conditional**
Finishing time	1103796	919280	818585
$k = 2$			
Finishing time	1655694	1286662	1085272

Figure 5.5: Minimal finishing times for *MP3-decoder*.

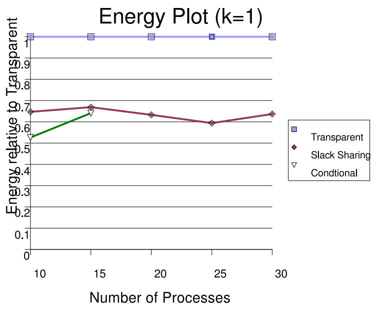
better. This tendency is even more obvious for $k = 2$, where the slack sharing scheduler performs 20% better, the conditional scheduler an amazing 50% than the transparent scheduler.

To see how large energy savings can be achieved by exploiting the slack produced by the better performing schedulers, we now minimise the energy for the designs. The deadline used is that of the fastest slack sharing schedule, and no reliability goal is set. The results of this experiment is shown in figure 5.6. For the energy plot ,the y-axis is the energy consumption relative to that of the fastest fully transparent schedule, calculated similarly to the finishing times in the previous experiment. For the reliability plot, the y-axis is the absolute reliability.

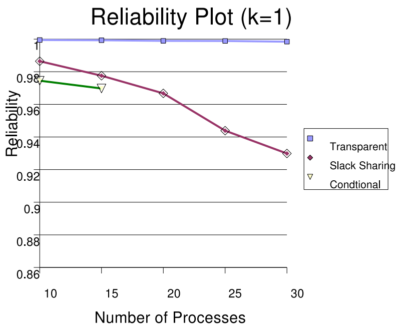
We see that the slack sharing schedule gives a dynamic energy saving of 30% and 45-50% for one and two faults respectively. The conditional schedule produces schedules that saves as much as a 70% for $k = 2$. However, we see that the conditional scheduling is not able to produce any results at all for graph sizes larger than 15. This is because of the use of *FT-CPG*s to capture all possible fault scenarios. The size of a *FT-CPG* grows drastically, with growing graph sizes. For graphs with 15 processes the average number of processes in the corresponding *FT-CPG*, is 70.9 for $k = 1$ and 270.9 for $k = 2$. With 20 processes this increases to 104.9, and 483.4 respectively, which renders the search infeasible.

The reliability plots show the energy savings are obtained at a very high reliability cost.

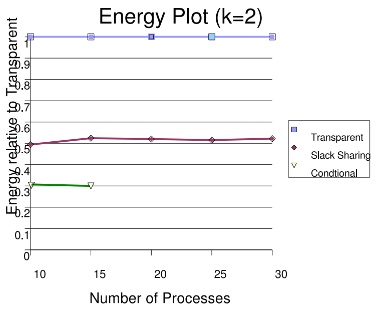
The finishing time optimisation results for the *MP3-decoder* are shown in figure 5.5. For $k = 2$ the slack sharing schedule is about 25% faster, and the conditional about 30% faster. This is comparable to the synthetic applications. The finishing times achieved using the fully transparent scheduling scheme, are used as deadlines in the remaining experiments. Recalling that the application has a deadline of $25ms$, the processor that runs the application, will have to execute



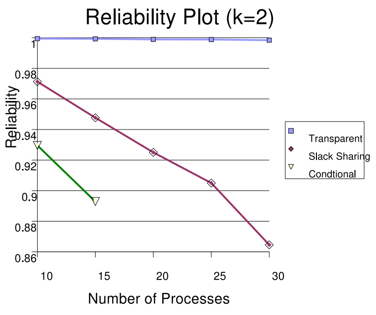
(a) Energy for $k = 1$



(b) Reliability for $k = 1$



(c) Energy $k = 2$



(d) Reliability $k = 2$

	No. of Processes				
No. Faults	10	15	20	25	30
Fully transparent					
$k = 1$	10/0/0	8/2/0	8/1/1	4/3/3	0/6/4
$k = 2$	10/0/0	10/0/0	10/0/0	6/2/2	0/7/3
Slack sharing					
$k = 1$	9/0/1	9/1/0	2/6/2	3/3/4	0/2/8
$k = 2$	9/1/0	10/0/0	3/5/2	2/6/2	0/4/6
Conditional					
$k = 1$	4/1/5	0/3/7	0/0/10	0/0/10	0/0/10
$k = 2$	5/0/5	1/0/9	0/0/10	0/0/10	0/0/10

(e) Finishing status of searches

Figure 5.6: Comparison of the obtainable energy savings for different schedulers. The fastest schedule for fully transparent scheduling has been used as deadline, and energy minimised under this. The energies are relative to the energy for running all processes at full speed, hence for energy smaller values are better. The reliability plots are in absolute values, and higher values are better.

	Fault Tolerance Technique		
$k = 1$	**Fully Trans.**	**Slack Shr.**	**Conditional**
Energy	46123.2084	24529.21914375	20598.871287222
Reliability	9 nines and 6	8 nines and 6	8 nines and 6
$k = 2$			
Energy	46123.2084	15234.397575	14606.184732684
Reliability	14 nines and 5	12 nines and 9	12 nines and 8

Figure 5.7: Energy and reliability for *MP3-decoder*.

at a minimum of:

$$f_{k=1} = \frac{1103796}{25ms} = 44.15Mhz \tag{5.2}$$

for $k = 1$ to finish within the deadline. For the implementation tolerating two fault, the minimum clock frequency is:

$$f_{k=2} = \frac{1655694}{25ms} = 66.23Mhz \tag{5.3}$$

Provided an architecture is given, which supports the two levels of fault tolerance using the straightforward fully transparent scheduling, the use of the more advanced schedulers could provide the same level of fault-tolerance while consuming the energy shown in figure 5.7. Both slack sharing and conditional scheduling give an energy saving of about 65% for $k = 2$.

The experiments show that significant savings in terms of slack, and energy, are available by the use of more sophisticated fault tolerant scheduling. The energy minimisation experiment clearly illustrates that energy management should be used with care, as the reliability of the produced schedules drops rapidly as the graphs sizes increase (i.e. the amount of slack increases). The experiments further show, that the conditional scheduling, although it produces very good results, is impractically slow.

5.5 List Scheduling vs. Optimal Schedules

To evaluate the efficiency of the proposed scheduling techniques, I have compared their performance against the algorithm presented in [10]. I have used a version of the original code, which schedules an *FT-CPG* representation of a graph using *list scheduling*. In these experiments 100% of the processes are considered to be recovered in case of fault.

The graphs are scheduled on an architecture with 4 processing units. The same mapping is used for all experiments. The optimisation criterion is finishing time.

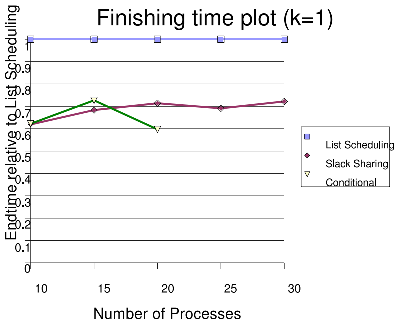
The results of the comparison with the list scheduling are shown in figure 5.8. In the plots the y-axis is the finishing time relative to that of the list scheduler. We see that the slack sharing scheduler performs 25% better for $k = 1$. For $k = 2$ the slack sharing approach still produces better results, but somewhat less so (about 20%). Again we see that the conditional scheduling, does not produce any results for larger graphs. For the schedules found, however, we see that the list scheduling results, are far from optimal. In fact, for $k = 2$ the optimal schedules are as much as 60% better.

The results for the *MP3-decoder*are shown in figure 5.7. For the case study, we see that the slack sharing scheduling is performing nearly as good as the conditional. We also note that the energy consumption is indeed lower for the schedules that handle two faults. This is because they have been scheduled with a different deadline. We recall that the deadline is set to the length of the fastest transparent schedule. The advantage of using more advanced scheduling algorithms over the transparent becomes more apparent as k increases, and hence there is more slack for voltage scaling.

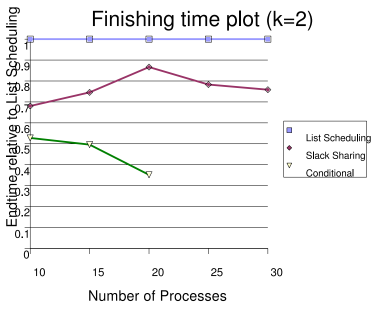
These experiments show that the proposed slack sharing scheduling performs significantly better than the list scheduling algorithm proposed in [10].

5.6 The Effects of Policy Assignment

The impact of policy assignment and mapping on the quality of obtainable schedules is evaluated by scheduling the same graph with three different degrees of mapping. Firstly, with all processes mapped, and all recovery executions mapped on the same processing element, i.e. only with re-execution. Secondly, with all root processes mapped, but recovery executions unmapped, i.e. combination of re-execution and passive replication. And finally with all processes unmapped. These experiments are run on the architecture shown in figure 5.9. This is a heterogeneous architecture, with three processors, each with different performance. The performance ratios are written inside the processing elements in the figure. The ratios mean, that a process mapped on PE_2 will have a duration that is twice as long as if it were mapped on PE_1. For this experiment



(a) $k = 1$



(b) $k = 2$

	No. of Processes				
No. Faults	10	15	20	25	30
Slack sharing					
$k = 1$	10/0/0	7/1/2	5/3/2	7/0/3	4/0/6
$k = 2$	10/0/0	6/1/3	6/1/2	10/0/0	8/0/2
Conditional					
$k = 1$	7/3/0	0/10/0	0/10/0	0/0/10	0/0/10
$k = 2$	4/6/0	0/10/0	0/1 /10	0/0/10	0/0/10

(c) Finishing status of searches

Figure 5.8: Fastest possible schedules with three different schedulers. The heuristic list scheduling approach is used for reference, and all other numbers are relative to this. Smaller values are better.

the optimisation criteria is finishing time.

The results of these experiments are shown in figure 5.10 for the synthetic applications. The graphs show that using a combination of re-execution and passive replication, my implementation can produce schedules which are consistently 10% better than those with only re-execution. We also see that if the optimisation tool is allowed to determine the mapping, as well as the policy assignment, the results become even better. The plot for this case, however behaves strangely, and the results become increasingly bad, and for 30 processes, even produces schedules that are worse than those with re-execution only. This is because the size of this design space being significantly larger in this case. In turn, this is seen in the status table, where nearly no searches finish within the timeout. Hence the schedules plotted are intermediate results and are thus sub-optimal. If the searches had finished, an improvement in the finishing time, similar to that where $k = 1$ should be expected.

The *MP3-decoder* is scheduled on the same, homogeneous, architecture as in

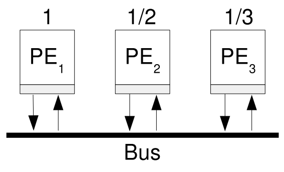
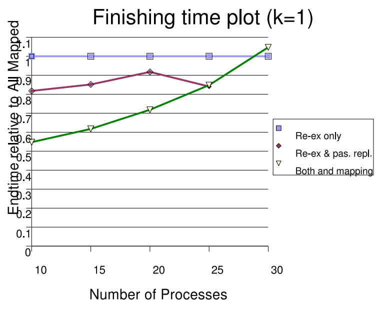
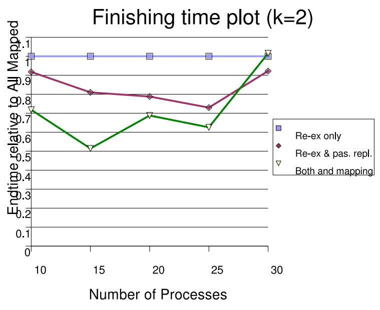


Figure 5.9: Architecture for the experiments in section 5.6. The performance ratios for the processors are written above each processor.



(a) $k = 1$



(b) $k = 2$

	No. of Processes				
No. Faults	10	15	20	25	30
Only re-execution					
$k = 1$	10/0/0	6/2/2	10/0/0	8/2/0	2/0/8
$k = 2$	10/0/0	10/0/0	8/0/2	10/0/0	8/0/2
Re-execution and passive replication					
$k = 1$	10/0/0	6/4/0	6/2/2	8/2/0	0/0/10
$k = 2$	10/0/0	6/4/0	8/0/2	8/2/0	0/6/4
Both and mapping					
$k = 1$	6/4/0	0/8/2	0/10/0	0/10/0	0/2/8
$k = 2$	4/6/0	0/10/0	0/8/2	0/10/0	0/8/2

(c) Finishing status of searches

Figure 5.10: Influence of policy assignment on quality of solutions. The schedules with all processes mapped are used as reference, and the other values are relative to this. Smaller values are better.

	Fault-tolerance policy		
$k = 1$	**Re-ex.**	**Re-ex. & pas. repl.**	**Both & mapping**
Finishing Time	919280	896671	835325
$k = 2$			
Finishing Time	1286662	1241444	1118752

Figure 5.11: Influence of policy assignment on schedule quality for *MP3-decoder*. The results are for: all processes mapped, only the root processes mapped, and all processes unmapped.

the other experiments. The results are shown in figure 5.11. Again, we see an improvement when using both passive replication and re-execution, and an even bigger improvement if the mapping is also considered part of the optimisation. The finishing times are improved by a few percent when passive replication is introduced, and by about 10% when mapping is also decided. Considering that the *MP3-decoder* has a highly parallel structure, and is scheduled on two identical processors, these improvements are in fact quite high.

These experiments show that considering policy assignment and mapping is critical to produce schedules of high quality.

5.7 Energy Trade-Offs for Reliability

In this experiment the obtainable energy savings possible under a reliability goal R_g have been investigated. The optimisation is done using slack sharing scheduling, which, through the previous examples, has been shown to behave well, both in terms of the quality of the produced schedules, and also in terms of execution time.

Two energy optimisations are done for each graph, one where the reliability is not constrained, and one where the reliability goal R_g is imposed as hard constraint. The imposed reliability goal is the one presented in section 5.3. The energy is compared to that of the fastest transparent schedule, the finishing time of which, is used as deadline.

The results of these experiments are shown in figure 5.12. The plots clearly show, that lowering the voltage to minimise energy consumption, without concern for reliability produces extremely unreliable systems. The reliability decreases dramatically with increasing application size and k. If, however, reliability is constrained to meet a reliability design goal, this is avoided. The plots for energy show that, the difference in energy is only very little, however the reliability benefits greatly from the introduced reliability goal. This shows that considering reliability as part of the optimisation process, my approach is able to dramatically improve the reliability of designed systems at very little sacrifice of energy.

To illustrate this, let us consider the probability of error, for $k = 1$ and 20 processes. This is improved by a factor of:

$$\Delta\rho = \frac{1 - R_{unconstrained}}{1 - R_{constrained}} = \frac{1 - 0.9634}{1 - 0.9906} = 3.9150 \tag{5.4}$$

At the cost of an increase in energy consumption by a mere:

$$\Delta E = \frac{E_{constrained} - E_{unconstrained}}{E_{unconstrained}} = \frac{0.6993 - 0.6917}{0.6917} = 0.88\% \tag{5.5}$$

And the trend is only more obvious if examples for $k = 2$ are considered.

The results for the *MP3-decoder* show the same tendency. Firstly, it should be noted that the reliability for this example is far greater than that of the random graphs. This is due to all processes being redundant. In the synthetic examples only 50% are made redundant, and hence the other half will contribute to the systems unreliability.

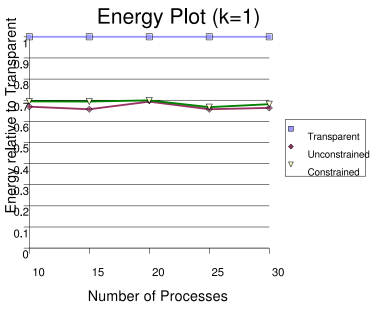
We see that, for $k = 1$, the unconstrained schedule consumes:

$$\Delta E = \frac{E_{unconstrained}}{E_{transparent}} = \frac{24529.21914375}{46123.2084} = 53.2\% \tag{5.6}$$

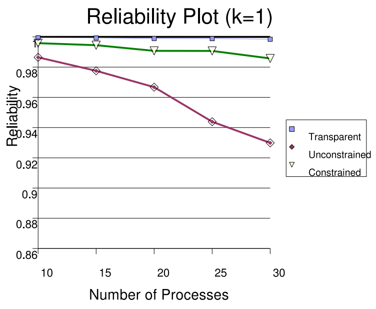
of the energy of the transparent schedule. However, the reliability is missed. The constrained schedule yields a schedule which consumes:

$$\Delta E = \frac{E_{constrained}}{E_{transparent}} = \frac{29020.2215625}{46123.2084} = 62.9\% \tag{5.7}$$

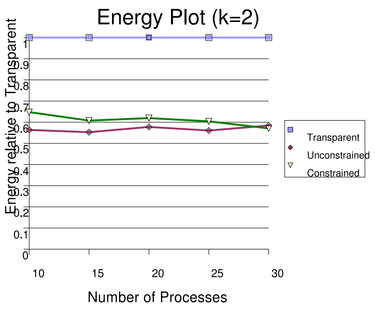
energy and meets the reliability goal. By sacrificing an energy saving of 9%, we have made the designed system meet its reliability goal. The energy cost, for achieving this, in this case is a bit larger than for the synthetic applications.



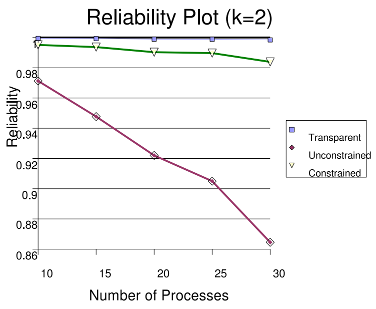
(a) Energy for $k = 1$



(b) Reliability for $k = 1$



(c) Energy for $k = 2$



(d) Reliability for $k = 2$

	No. of Processes				
No. Faults	10	15	20	25	30
Fully transparent					
$k = 1$	10/0/0	9/0/1	7/3/0	5/4/1	2/6/2
$k = 2$	10/0/0	10/0/0	7/3/0	6/3/1	3/5/2
Unconstrained					
$k = 1$	9/1/0	9/1/0	4/5/1	6/3/1	0/5/5
$k = 2$	9/1/0	10/0/0	4/6/0	4/5/1	0/6/4
Constrained					
$k = 1$	9/1/0	9/0/1	4/3/3	6/3/1	0/7/3
$k = 2$	10/0/0	10/0/0	5/4/1	7/1/2	0/6/4

(e) Finishing status of searches

Figure 5.12: Plot of energy and relibality, scheduled with and without constrained reliability.

	Transparent	**Unconstrained**	**Constrained**
$k = 1$			R_g = *8 nines and 8*
Energy	46123.2084	24529.21914375	29020.2215625
Reliability	9 nines and 5	8 nines and 6	8 nines and 9
$k = 2$			R_g = *13 nines and 6*
Energy	46123.2084	15234.39757	18931.3788375
Reliability	14 nines and 6	12 nines and 8	13 nines and 6

Figure 5.13: Energy and reliability for *MP3-decoder*.

This is because the *MP3-decoder* is dominated by a few very heavy processes, as discussed earlier. This makes the application have little flexibility to do voltage scaling. The synthetic applications have processes of more even sizes, and hence are more flexible.

This demonstrates that reliability should be considered as a part of the system-level optimisation process. Doing so, may yield valuable insight on how to efficiently voltage scale a system. It is possible to achieve much better reliability, at the cost of only a very little increase in consumed energy.

Chapter 6

Conclusions

In this thesis, I present design optimisation approaches for the design of time constrained fault-tolerant embedded multiprocessor systems-on-a-chip. The presented techniques consider the reliability simultaneously with the scheduling, mapping and voltage scaling. The presented approaches are able to produce schedules with good reliability at a very small energy cost, compared to schedules that where scheduled without considering reliability. This shows that it is critical to consider the reliability of systems as part of the system-level design phase.

To evaluate the reliability of fault tolerant systems, I have derived equations for the reliability of several different fault-tolerance techniques. This extends the work of [30] to allow for not only re-execution, but also replication and passive replication. Further the expressions are generalised for arbitrary numbers of handled faults k, and not as previous work only for $k = 1$.

Three different fault tolerant scheduling methods have been implemented: fully transparent, slack sharing and conditional. To do conditional scheduling, a general algorithm has been developed which builds *FT-CPG*s from normal task graphs. This extends the work of [9] and [10].

The approaches have been implemented using a constraint logic programming system, and towards this, end the logic constraints to model fault-tolerant embedded systems are presented. The advantage of using an *CLP*approach is its flexibility. It is easy to add and remove constraints, and several design tasks can be integrated in the same code. The search strategy can easily be controlled, and, given enough time, the search is able to find optimal solutions.

The experiments conducted have shown that the presented algorithms are able to produce implementations which are fault-tolerant, schedulable, and minimise energy.

6.1 Further Work

The model presented disregards communication delay. Considering this however will give rise to some interesting problems. For the more advanced scheduler implementation, fault information has to be shared between processors. This information will naturally have to be transmitted on the bus. This may lead to congestion on the bus, and hence impact execution speed. Further, the extra power needed to drive the bus, may affect the optimal schedules, such that less parallelism is favoured.

The optimisation tool presented in this thesis does complete optimal search. This is shown to yield very good results, but is also rather slow. Comparison with a list scheduling heuristic shows that the implemented tool behaves generally between 10-20 % better for slack sharing scheduling. It would be interesting to extend the presented model with a fast constraint logic programming search heuristic, which would quickly produce solutions of good quality.

Appendix A

Derivation of *FT-CPG*

In [9] an algorithm to derive *FT-CPG*s from normal process graphs is presented. It is however very abstract, and as such does not form a good base for doing an actual implementation. As a consequence we have proposed a directly implementable algorithm to derive *FT-CPG*s.

The algorithm takes as input a normal directed process graph, and from it produces a conditional graph with fault tolerance for k faults. An example of a process graph and its derived *FT-CPG* for $k = 1$ is shown in figure 2.6.

The GenerateFTCPG function is shown in pseudo-code representation in algorithm 1. The *subscripts* array set up in line 3 is used to assign unique subscript numbers to re-executions of a process. This is necessary to distinguish re-executions from each other. Lines 1 to 22 initialise the data structures. The algorithm starts with the source node, which has no predecessors. The algorithm maintains a set of conditions for each process which captures the fault scenario in which the process will be active. For each process with predecessors, it is necessary to determine which instances of this/these predecessors are valid. E.g. it is critical that a process with two predecessors, is only inserted into the graph, with combinations of these two predecessors, that do not belong to mutually exclusive paths in the *FT-CPG*. A combination of predecessor processes that is part of the same path in the *FT-CPG*, is called a valid combination. These valid combinations are determined using the set of conditions for each predecessor. The valid combinations are determined in line 25. To find these valid combinations is non-trivial, and the algorithm for doing this is given in algorithm 2.

The *for* loop from line 26 to 33 inserts a new process for each valid combination, and the loop from line 34 to 45 inserts re-executions of each of the newly inserted processes.

A.1 FindValidCombinations

This is the function to find valid combinations of predecessors for a process. The function takes as input the so far generated *FT-CPG* and the process for which we wish find predecessor combinations. The process will have a set of predecessors in its original graph, in figure 2.5 P_4 has the predecessors P_2 and P_3. The loop in lines 3-5 searches through the *FT-CPG* generated so far, and inserts each instance of the predecessors processes in the parents array. In line 6 the function CompareConds is called. This will determine which combinations, of the found predecessors in the parents array, are part of the same path, and hence are valid candidates for having P_i inserted as a child. Lines 7 to 11 ensures that none of the valid combinations are paths with more than k faults.

A.2 CompareConds

This function takes a set of lists, holding all the instances of the predecessor processes for a process. The predecessors are sorted, such that instances of the same process are all in a separate list. Lines 2-3 handles the special case when there is only one parent, in which case all instances are valid candidates. Line 4 extracts the first set of predecessors, calls CompareConds recursively with the remaining predecessors. This recursion merges the process sets, such that processes that are part of the same path through the graph are joined, and their conditions joined as well. The function returns a single set of combinations of predecessor processes that are parts of the same paths. Using the example from figure 2.6 and considering process P_4, the function would be called with the predecessor lists $\{P_{2,1}, P_{2,2}, P_{2,3}\}$ and $\{P_{3,1}, P_{3,2}, P_{3,3}\}$, and would return the list $\{\{P_{2,1}, P_{3,1}\}, \{P_{2,1}, P_{3,3}\}, \{P_{2,3}, P_{3,1}\}, \{P_{2,2}, P_{3,2}\}\}$.

The function CompareConds evaluates the conditions for all sets of parents and see if they are part of mutually exclusive paths in the graph. The algorithm for this function is shown in algorithm 3.

Algorithm 1 GenerateFTCPG($\mathcal{G}$,$\mathcal{T}$,k)

```
 1: $G \leftarrow \emptyset$
 2: $ReadyList \leftarrow \emptyset$
 3: $subscript$ [number of processes]
 4: set all $susbcript$ to 1
 5: $P_i \leftarrow SourceNode(\mathcal{G})$
 6: $subscript \leftarrow subscripts[i] + +$
 7: Insert($P_{i,1}$, $\mathcal{G}$)
 8: $tmp \leftarrow P_{i,subscript}$
 9: $P_{i,1}.possibleFaults \leftarrow k$
10: $P_{i,1}.conditions \leftarrow \emptyset$
11: $tmp \leftarrow P_{i,1}$
12: for $j \leftarrow P_{i,1}.possibleFauls$ downto 0 do
13:    $subscript \leftarrow subscripts[i] + +$
14:    Insert($P_{i,subscript}$, $\mathcal{G}$)
15:    $P_{i,subscript}.possibleFaults \leftarrow j$
16:    $P_{i,subscript}.conditions \leftarrow tmp.conditions + tmp.fail$
17:    Connect($tmp$, $P_{i,subscript}$)
18:    $tmp \leftarrow P_{i,subscript}$
19: end for
20: $P_{i,1}.condition+ = P_{i,1}.success$
21: $newProcessess \leftarrow \emptyset$
22: Insert($P_i.children$, $readyList$)
23: while $ReadyList$ is not empty do
24:    $P_i \leftarrow$ ExtractFirst($ReadyList$)
25:    $VC \leftarrow$ FindValidCombinations($P_i, G, k$)
26:    for all $vc \in VC$ do
27:       $subscript \leftarrow subscripts[i] + +$
28:       Insert($P_{i,subscript}$, $\mathcal{G}$)
29:       Insert($P_{i,subscript}$, $newProcesses$)
30:       $P_{i,subscript}.possibleFaults \leftarrow k - vc.faults$
31:       $P_{i,subscript}.conditions \leftarrow vc.conditions$
32:       Connect($vc.processes$, $P_{i,subscript}$)
33:    end for
34:    for all $P_{i,k} \in newProcesses$ do
35:       $tmp \leftarrow P_{i,k}$
36:       for $j \leftarrow tmp.possibleFauls$ downto 0 do
37:          $subscript \leftarrow subscripts[i] + +$
38:          Insert($P_{i,subscript}$, $\mathcal{G}$)
39:          $P_{i,subscript}.possibleFaults \leftarrow tmp.possibleFaults$
40:          $P_{i,subscript}.conditions \leftarrow tmp.conditions + tmp.fail$
41:          Connect($tmp$, $P_{i,subscript}$)
42:          $tmp \leftarrow P_{i,subscript}$
43:       end for
44:       $P_{i,k}.condition+ = P_{i,k}.success$
45:    end for
46:    Insert($P_i.children$, $readyList$)
47: end while
```

Algorithm 2 FindValidCombinations(P_i, G, k)

```
1: $parents \leftarrow \emptyset$
2: $index \leftarrow 1$
3: for all $P_j \in P_i.parents$ do
4:   $parents[index++] \leftarrow$ all instances of $P_j \in G$
5: end for
6: $candidates \leftarrow$ CompareConds($parents$)
7: for all $candidate \in candidates$ do
8:   if $candidate.errors > k$ then
9:     Remove($candidate, candidates$) {Ensure that only process with less than $k$ faults are added}
10:  end if
11: end for
12: return $candidates$
```

Algorithm 3 CompareConds($parents$)

```
1: $VC \leftarrow \emptyset$
2: if $parents.size = 1$ then
3:   $VC.process+ = parents$ {There is only one set of processes in the set}
4: else
5:   $head \leftarrow$ ExtractFirst($parents$) {Extract first set}
6:   $rest \leftarrow$ CompareConds($parents$) {And recurse}
7:   for all $h \in head$ do
8:     for all $r \in rest$ do
9:       $valid \leftarrow$ true
10:      for all $a \in h.conditions$ do
11:        for all $b \in r.conditions$ do
12:          if $a.process = b.process$ and $a.value! = b.value$ then
13:            $valid \leftarrow$ false {The processes are dependent on different conditions, and do hence not form a valid combination}
14:          end if
15:        end for
16:      end for
17:      if $valid = true$ then
18:        Insert($h + r, VC$) {Merge the conditions and process of $h$ and $r$ into $VC$}
19:      end if
20:    end for
21:  end for
22: end if
23: return VC
```

A.3 Replication

Also replication can be captured by a *FT-CPG*. This is illustrated in figure 2.6(c) where P_1 is replicated. Each vertex in the internal graph representation has a number which describes the amount of replicas it has. As all replicas must necessarily have the same in and outbound edges, only a single vertex is used to model replication internally. When outputting the graph, this vertex is simply output the same amount of times as it has replicas (with unique subscripts). In this way replication can be handled simply and elegantly using the same algorithm as for re-execution.

Bibliography

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

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

[3] Cristian Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23:14–19, 2003.

[4] R. Dick, D. Rhodes, and W. Wolf. Tgff: Task graphs for free. In *Int. Workshop Hardware/Software Codesign*, pages 97–101, March 1998.

[5] ECL^{i}PS^e project homepage. Web Page. `http://eclipse-clp.org/`.

[6] A. Ejlali, B. M. Al-Hashimi, M. Schmitz, P. Rossing, and S. G. Miremadi. Combined time and information redundancy for seu-tolerance in energy-efficient real-time systems. *IEEE Transactions on VLSI Systems*, 14(4):323–335, 2006.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 2003.

[8] Dr. Randy Isaac. Influence of technology directions on system architecture. Presentation, September 2001. `research.ac.upc.edu/pact01/keynotes/isaac.pdf`.

[9] Viacheslav Izosimov. *Scheduling and Optimization of Fault-Tolerant Embedded Systems*. PhD thesis, Linköping University, 2006.

[10] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. Synthesis of fault-tolerant schedules with transparency/perofrmance trade-offs for distributed embedded systems.

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

[12] Nagarajan Kandasamy, John P. Hayes, and Brian T. Murray. Transparent recovery from intermittent faults in time-triggered distributed systems. *IEEE Transactions on Computers*, 52(2), February 2003.

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

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

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

[16] W. Kuo and V. R. Prasad. An annotated overview of system-reliability optimization. *IEEE Transactions on Reliability*, 49:176–187, 2000.

[17] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[18] Jan Madsen, Kashif Virk, and Mercury Jair Gonzalez. *A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor Systsems-on-Chips*, chapter 10. Morgan-Kaufmann Publishers, 2004. in book: *Multiprocessor Systems-on-Chip.*

[19] Steven M. Martin, Krisztian Flautner, Trevor Mudge, and David Blauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. pages 721–725, 2002.

[20] Rami Melhem, Daniel Mossé, and Elmootazbellah Elnozahy. The interplay of power management and fault recovery in real-time systems. In *IEEE Transactions on Computers*, 2003.

[21] A. Orailoglu and R. Karri. Coactive scheduling and checkpoint determination during high level synthesis of self-recovering microarchitectures. *IEEE Transactions on VLSI Systems*, pages 304–311, 1994.

[22] 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 *Proceedings of the Design Automation and Test in Europe Conference*, pages 1164–1169, 2004.

[23] Paul Pop, Petru Eles, and Zebo Peng. *Analysis and Synthesis of Communication-Intensive Heterogenous Real-Time Systems.* Kluwer Academic Publishers, 2004.

[24] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems.* Kluwer Academic Publishers, 2004.

[25] Martin L. Shooman. *Reliability of Computer Systems and Networks.* Wiley, 2001.

[26] *TGFF* project homepage. Web Page. http://ziyang.ece.northwestern.edu/tgff/.

[27] The international technology roadmap for semiconductors, 2005. http://www.itrs.net/Links/2005ITRS/SysDrivers2005.pdf.

[28] The international technology roadmap for semiconductors, 2006 update, 2006. http://www.itrs.net/Links/2006Update/FinalToPost/01_SysDrivers_2006UPDAT

[29] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal Computer Systems Science*, 10(3):384–393, 1975.

[30] Dakai Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[31] Dakai Zhu, Rami Melhem, and Daneil Mossé. Analysis of an energy efficient optimistic tmr scheme. In *Proc. of the 10th International Conference on Parallel and Distributed Systems*, July 2004.

[32] Dakai Zhu, Rami Melhem, and Daniel Mossé. The effects of energy management on reliability in real-time embedded systems. *ICCAD*, 2004.