Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour

Sorin Manolache
Linköping

Abstract

Embedded systems have become indispensable in our life: household appliances, cars, airplanes, power plant control systems, medical equipment, telecommunication systems, space technology, they all contain digital computing systems with dedicated functionality. Most of them, if not all, are real-time systems, i.e. their responses to stimuli have timeliness constraints.

The timeliness requirement has to be met despite some unpredictable, stochastic behaviour of the system. In this thesis, we address two causes of such stochastic behaviour: the application and platform-dependent stochastic task execution times, and the platform-dependent occurrence of transient faults on network links in networks-on-chip.

We present three approaches to the analysis of the deadline miss ratio of applications with stochastic task execution times. Each of the three approaches fits best to a different context. The first approach is an exact one and is efficiently applicable to monoprocessor systems. The second approach is an approximate one, which allows for designer-controlled trade-off between analysis accuracy and analysis speed. It is efficiently applicable to multiprocessor systems. The third approach is less accurate but sufficiently fast in order to be placed inside optimisation loops. Based on the last approach, we propose a heuristic for task mapping and priority assignment for deadline miss ratio minimisation.

Our contribution is manifold in the area of buffer and time constrained communication along unreliable on-chip links. First, we introduce the concept of communication supports, an intelligent combination between spatially and temporally redundant communication. We provide a method for constructing a sufficiently varied pool of alternative communication supports for each message. Second, we propose a heuristic for exploring the space of communication support candidates such that the task response times are minimised. The resulting time slack can be exploited by means of voltage and/or frequency scaling for communication energy reduction. Third, we introduce an algorithm for the worst-case analysis of the buffer space demand of applications implemented on networks-on-chip. Last, we propose an algorithm for communication mapping and packet timing for buffer space demand minimisation.

All our contributions are supported by sets of experimental results obtained from both synthetic and real-world applications of industrial size.

This work has been supported by ARTES (A Network for Real-Time Research and Graduate Education in Sweden) and STRINGENT

(Strategic Integrated Electronic Systems Research at Linköpings Universitet).

Acknowledgements

This work would not have been possible without the contribution of many people, to whom I would now take the opportunity to thank.

Foremost, I thank Prof. Petru Eles, my considerate adviser. His confidence in our work compelled me to persist, while his personality commends my admiration. Prof. Zebo Peng has been the ideal research group director during these years. I thank him for his soft and efficient leading style.

The embedded systems laboratory, and in a bigger context the whole department, provided an excellent environment for professional and personal development. I thank all those who contributed to it, foremost my colleagues in ESLAB for their friendship, spirit, and time.

Some of the work in this thesis stems from the period in which I visited the research department of Ericsson Radio Systems lead by Dr. Peter Olanders. I want to thank him, as well as Dr. Béatrice Philibert and Erik Stoy for providing me that opportunity.

New interests and enthusiasm, professional but also cultural, were awoken in me during my visit to the group led by Prof. Radu Mărculescu at the Carnegie-Mellon University. I take this opportunity to thank him for inviting me and to salute the people I met there.

Geographically more or less distant, some persons honoured me with their friendship for already frighteningly many years. It means very much to me.

I am grateful to my mother for the mindset and values she has passed to me. In the same breath, I thank my father for the smile and confidence he has always offered me.

Last, I thank Andrea for filling our lives with joy.

Sorin Manolache
Linköping, October 2005

Contents

Part I

Preliminaries

Chapter 1

Introduction

This chapter briefly presents the frame of this thesis, namely the area of embedded real-time systems. It introduces the two aspects of stochastic behaviour of real-time systems that we address in this thesis, namely the application-specific and platform-specific stochastic task execution times and the platform-specific transient faults of hardware. The chapter summarises the contributions and draws the outline of the thesis.

1.1 Embedded System Design Flow

Systems controlled by embedded computers become indispensable in our lives and can be found in avionics, automotive industry, home appliances, medicine, telecommunication industry, mecatronics, space industry, etc. [Ern98].

Very often, these embedded systems are reactive, i.e. they are in steady interaction with their environment, acting in a prescribed way as response to stimuli received from the environment. In most cases, this response has to arrive at a certain time moment or within a prescribed time interval from the moment of the application of the stimulus. Such systems, in which the correctness of their operation is defined not only in terms of functionality but also in terms of timeliness, form the class of real-time systems [But97, KS97, Kop97, BW94].

Timeliness requirements may be *hard* meaning that the violation of any such requirement is not tolerated. In a hard real-time system, if not all deadlines are guaranteed to be met, the system is said to be unschedulable. Typical hard real-time application domains are plant control, aircraft control, medical, and automotive applications. Systems classified as *soft* real-time may occasionally break a real-time requirement provided that the service quality exceeds prescribed levels.

The nature of real-time embedded systems is typically heterogeneous along multiple dimensions. For example, an application may exhibit data, control and protocol processing characteristics. It may also consist of blocks exhibiting different categories of timeliness requirements, such as hard and soft. Another dimension of heterogeneity is given by the environment the system operates in. For example, the stimuli and responses may be of both discrete and continuous nature.

The heterogeneity in the nature of the application itself on one side and, on the other side, constraints such as cost, performance, power dis-

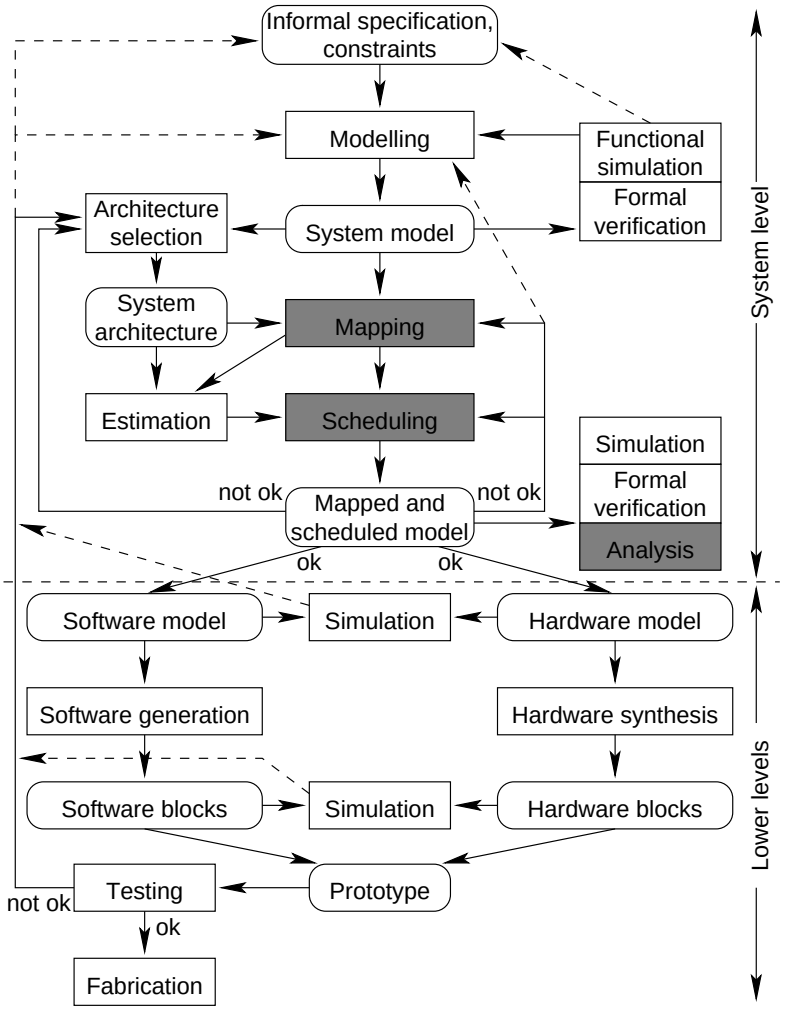


Figure 1.1: Typical design flow

sipation, legacy designs and implementations, as well as requirements such as reliability, availability, security, and safety, often lead to implementations consisting of heterogeneous multiprocessor platforms.

Designing such systems implies the deployment of different techniques with roots in system engineering, software engineering, computer architectures, specification languages, formal methods, real-time scheduling, simulation, programming languages, compilation, hardware synthesis, etc. Considering the huge complexity of such a design task, there is an urgent need for automatic tools for design, estimation, and synthesis in order to support and guide the designer. A rigorous, disciplined, and systematic approach to real-time embedded system design is the only way the designer can cope with the complexity of current and future designs in the context of high time-to-market pressure. A simplified view of such a design flow is depicted in Figure 1.1 [Ele02].

The design process starts from a less formal specification together with a set of constraints. This initial informal specification is then captured as a more rigorous model formulated in one or possibly several

modelling languages [JMEP00]. Next, a system architecture is chosen as the hardware platform executing the application. This system architecture typically consists of programmable processors of various kinds (application specific instruction processors (ASIPs), general purpose processors, digital signal processors (DSPs), protocol processors), and dedicated hardware processors (application specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs)) interconnected by means of shared buses, point-to-point links or networks of various types. Once a system architecture is chosen, the functionality, clustered in tasks or pieces of functionality of certain conceptual unity, are *mapped* onto (or assigned to) the processors or circuits of the system architecture. The communication is mapped on the buses, links, or networks. Next, the tasks or messages that share the same processing element or bus/link are *scheduled*. The resulting mapped and scheduled system model is then estimated by means of analysis or simulation or formal verification or combinations thereof. During the system level design space exploration phase, different architecture, mapping, and scheduling alternatives are assessed in order to meet the design requirements and possibly optimise certain indicators. Once a design is found that satisfies the functional and non-functional requirements, the system is synthesized in the lower design phases.

At each design phase, the designers are confronted with various levels of uncertainty. For example, the execution times of tasks are unknown before the functionality is mapped to a system architecture. Even after mapping, a degree of uncertainty persists regarding the task execution times. For example, the state of the various levels of the memory hierarchy, which affects the data access latency and execution time, depends on the task scheduling. After a task scheduling has been decided upon, the task execution times heavily depend on the input data of the application, which in turn may be unpredictable. The task execution time is one of the components that induce a stochastic behaviour on systems.

Transient failures of various hardware components also induce a stochastic behaviour on the system. They may be caused by environmental factors, such as over-heating of certain parts of the circuit, or electro-magnetic interference, cross-talk between communication lines. Communication time and energy, and even application correctness may be affected by such phenomena.

We consider transient failures of on-chip communication links, while we assume that failures of other system components are tolerated by techniques outside the scope of this thesis.

There exists a need for design tools that take into account the stochastic behaviour of systems in order to support the designer. This thesis aims at providing such a support. We mainly address two sources that cause the system to have a stochastic behaviour: task execution times and transient faults of the links of on-chip networks.

1.2 Contribution

The shaded blocks in Figure 1.1 denote the phases of the design processes to which this thesis contributes.

With respect to the analysis of a mapped and scheduled model, we propose the following approaches:

1. Chapter 4 presents an approach that determines the exact deadline miss probability of tasks and task graphs of soft real-time applications with stochastic task execution times. The approach is efficient in the case of applications implemented on monoprocessor systems.
2. Chapter 5 presents an approach that determines an approximation of the deadline miss probability of systems under the same assumptions as those presented in Chapter 4. The approach allows a designer-controlled way to trade analysis speed for analysis accuracy and is efficient in the case of applications implemented on multiprocessor systems.
3. Chapter 6 presents an approach to the fast approximation of the deadline miss probability of soft real-time applications with stochastic task execution times. The analysis is efficiently applicable inside design optimisation loops.
4. Chapter 10 presents an approach to the analysis of the buffer space demand of applications implemented on networks-on-chip.

Our contribution to the functionality and communication mapping problem is the following:

1. Chapter 6 presents an approach to the task and communication mapping under constraints on deadline miss ratios.
2. Chapter 9 presents an approach to the communication mapping for applications implemented on networks-on-chip with unreliable on-chip communication links under timeliness, energy, and reliability constraints.
3. Chapter 10 presents an approach to the communication mapping for applications implemented on networks-on-chip with unreliable on-chip communication links under timeliness, buffer space, and reliability constraints.

Our contribution to the scheduling problem consists of an approach to priority assignment for tasks with stochastic task execution times under deadline miss ratio constraints. The approach is presented in Chapter 6.

The work presented in Chapter 4 is published in [MEP01, MEP04b]. Chapter 5 is based on [MEP02, MEP], and the contributions of Chapter 6 are published in [MEP04a]. The work presented in Chapter 9 is published in [MEP05] while Chapter 10 is based on [MEP06].

1.3 Thesis Organisation

The thesis is organised as follows.

Part II of this thesis deals with the stochastic behaviour caused by the non-deterministic nature of task execution. First, we present the motivation of our work in this area and we survey the related work (Chapter 2). Next, Chapter 3 introduces the notation and system model to be used throughout Part II. Then, Chapter 4 and Chapter 5 present two analytic performance estimation approaches, an exact one, efficiently applicable to monoprocessor systems, and an approximate analysis approach, efficiently applicable to multiprocessor systems. Part II concludes with an approach to the mapping of tasks on a multiprocessor platform and the priority assignment to tasks in order to optimise the deadline miss ratios.

Part III deals with the stochastic behaviour generated by the non-deterministic nature of transient faults occurring on links of on-chip networks. Chapter 7 introduces the context of the problem and surveys the related work. Chapter 8 describes the system model that we use throughout Part III. Chapter 9 introduces an approach to communication energy optimisation under timeliness and communication reliability constraints, while Chapter 10 presents an approach to buffer space demand minimisation of applications implemented on networks-on-chip.

Part IV concludes the thesis.

Part II

Stochastic Schedulability Analysis and Optimisation

Chapter 2

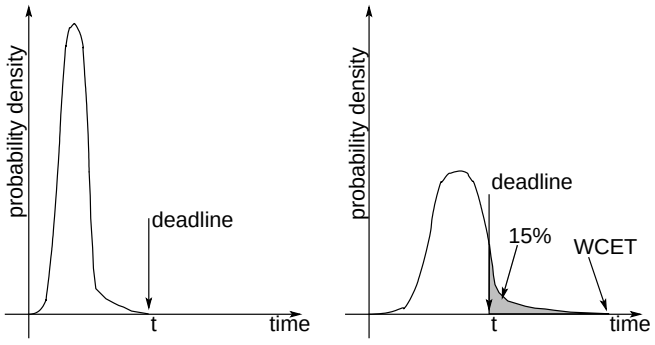
Motivation and Related Work

This chapter first motivates the work in the area of performance analysis of systems with stochastic task execution times. Next, related approaches are surveyed.

2.1 Motivation

Historically, real-time system research emerged from the need to understand, design, predict, and analyse safety critical applications such as plant control and aircraft control, to name a few. Therefore, the community focused on hard real-time systems, where breaking timeliness requirements is not tolerated. The analysis of such systems gives a yes/no answer to the question if the system fulfils the timeliness requirements. Hard real-time analysis relies on building worst-case scenarios. A scenario typically consists of a combination of task execution times. It is worst-case with respect to a timeliness requirement if either the requirement is broken in the given scenario or if the fact that the requirement is not broken in the given scenario implies that the system fulfils the requirement in all other possible scenarios. Hard real-time analysis cannot afford but to assume that worst-case scenarios always happen and to provision for these cases. This approach is the only one applicable for the class of safety critical embedded systems, even if very often leads to significant under-utilisation of resources.

For the class of soft real-time systems, however, such an approach misses the opportunity to create much cheaper products with low or no perceived service quality reduction. For example, multimedia applications like JPEG and MPEG encoding, sound encoding, etc. exhibit this property. In these situations, the designer may trade cost for quality. Thus, it is no longer sufficient to build worst-case scenarios, but it is more important to analyse the likelihood of each scenario. Instead of answering whether a timeliness requirement is fulfilled or not, soft real-time analysis answers questions such as what is the probability that the requirement is fulfilled or how often is it broken during the lifetime of the system. While in hard real-time analysis the tasks are assumed to execute for the amount of time that leads to the worst-case scenario, in soft real-time analysis task execution time probability dis-



(a) Fast and expensive processor

(b) Slower but inexpensive processor

Figure 2.1: Execution time probability density functions

tributions are preferred in order to be able to determine execution time combinations and their likelihoods.

The execution time of a task is a function of application dependent, platform dependent, and environment dependent factors. The amount of input data to be processed in each task instantiation as well as its type (pattern, configuration) are application dependent factors. The micro-architecture of the processing unit that executes a task is a platform dependent factor influencing the task execution time. If the time needed for communication with the environment (database lookups, for example) is to be considered as a part of the task execution time, then network load is an example of an environmental factor influencing the task execution time.

Input data amount and type may vary, as for example is the case for differently coded MPEG frames. Platform-dependent characteristics, like cache memory behaviour, pipeline stalls, write buffer queues, may also introduce a variation in the task execution time. Thus, obviously, all of the enumerated factors influencing the task execution time may vary. Therefore, a model considering the variability of execution times would be more realistic than the one considering just worst-case execution times. In the most general model, task execution times with arbitrary probability distribution functions are considered. These distributions can be extracted from performance models [van96] by means of analytic methods or simulation and profiling [van03, Gv00, Gau98]. Obviously, the worst-case task execution time model is a particular case of such a stochastic one.

Figure 2.1 shows two execution time probability density functions of the same task. The first corresponds to the case in which the task is mapped on a fast processor (Figure 2.1(a)). In this case, the worst-case execution time of the task is equal to its deadline. An approach based on a worst-case execution time model would implement the task on such an expensive processor in order to guarantee the imposed deadline for the worst-case situation. However, situations in which the task execution time is close to the worst-case execution time occur with small probability. If the nature of the application is such that a certain percentage of deadline misses is affordable, a cheaper system, which still

fulfils the imposed quality of service, can be designed. For example, on such a system the execution time probability density function of the same task could look as depicted in Figure 2.1(b). If it is acceptable for the task to miss 15% of its deadlines, such a system would be a viable and much cheaper alternative.

In the case of hard real-time systems, the question posed to the performance analysis process is whether the system is schedulable, which means if all deadlines are guaranteed to be met or not. In the case of soft real-time systems however, the analysis provides fitness estimates, such as measures of the degree to which a system is schedulable, rather than binary classifications. One such measure is the expected deadline miss ratio of each task or task graph and is the focus of this part of the thesis.

Performance estimation tools can be classified in simulation and analysis tools. Simulation tools are flexible, but there is always the danger that unwanted and extremely rare glitches in behaviour, possibly bringing the system to undesired states, are never observed. The probability of not observing such an existing behaviour can be decreased at the expense of increasing the simulation time. Analysis tools are more precise, but they usually rely on a mathematical formalisation, which is sometimes difficult to come up with or to understand by the designer. A further drawback of analysis tools is their often prohibitive running time due to the analysis complexity. A tool that trades, in a designer-controlled way, analysis complexity (in terms of analysis time and memory, for example) with analysis accuracy or the degree of insight that it provides, could be a viable solution to the performance estimation problem.

We aim at providing analytical support for the design of systems with stochastic task execution times. Chapters 4 and 5 present two approaches for the analysis of the deadline miss ratio of tasks, while Chapter 6 presents an approach for the minimisation of deadline miss ratios by means of task mapping and priority assignment.

2.2 Related Work

Before presenting our approach to the analysis and optimisation of systems with stochastic task execution times, we survey some of the related work in the area.

An impressive amount of work has been carried out in the area of schedulability analysis of applications with worst-case task execution times both for monoprocessor platforms [LL73, BBB01, LW82, LSD89, ABD+91, Bla76, ABRW93, SGL97, SS94, GKL91] and multiprocessor platforms [SL95, Sun97, Aud91, ABR+93, TC94, PG98] under fairly general assumptions.

Much fewer publications address the analysis of applications with stochastic task execution times. Moreover, most of them consider relatively restricted application classes, limiting their focus on monoprocessor systems, or on exponential task execution time probability distribution functions. Some approaches address specific scheduling policies or assume high-load systems.

Burns et al. [BPSW99] address the problem of a system breaking its timeliness requirements due to transient faults. In their case, the execution time variability stems from task re-executions. The short-

est interval between two fault occurrences such that no task exceeds its deadline is determined by sensitivity analysis. The probability that the system exceeds its deadline is given by the probability that faults occur at a faster rate than the tolerated one. Broster et al. [BBRN02] propose a different approach to the same problem. They determine the response time of a task given that it re-executes $k \in \mathbb{N}$ times due to faults. Then, in order to obtain the probability distribution of the response time, they compute the probability of the event that k faults occur. The fault occurrence process is assumed to be a Poisson process in both of the cited works. Burns et al. [BBB03] extend Broster's approach in order to take into account statistical dependencies among execution times. While their approaches are applicable to systems with sporadic tasks, they are unsuited for the determination of task deadline miss probabilities of tasks with generalised execution time probability distributions. Also their approaches are confined to sets of independent tasks implemented on monoprocessor systems.

Bernat et al. [BCP02] address a different problem. They determine the frequency with which a single task executes for a particular amount of time, called execution time profile. This is performed based on the execution time profiles of the basic blocks of the task. The strength of this approach is that they consider statistical dependencies among the execution time profiles of the basic blocks. However, their approach would be difficult to extend to the deadline miss ratio analysis of multi-task systems because of the complex interleaving that characterises the task executions in such environments. This would be even more difficult in the case of multiprocessor systems.

Atlas and Bestavros [AB98] extend the classical rate monotonic scheduling policy [LL73] with an admittance controller in order to handle tasks with stochastic execution times. They analyse the quality of service of the resulting schedule and its dependence on the admittance controller parameters. The approach is limited to monoprocessor systems, rate monotonic analysis and assumes the presence of an admission controller at run-time.

Abeni and Buttazzo's work [AB99] addresses both scheduling and performance analysis of tasks with stochastic parameters. Their focus is on how to schedule both hard and soft real-time tasks on the same processor, in such a way that the hard ones are not disturbed by ill-behaved soft tasks. The performance analysis method is used to assess their proposed scheduling policy (constant bandwidth server), and is restricted to the scope of their assumptions.

Tia et al. [TDS+95] assume a task model composed of independent tasks. Two methods for performance analysis are given. One of them is just an estimate and is demonstrated to be overly optimistic. In the second method, a soft task is transformed into a deterministic task and a sporadic one. The latter is executed only when the former exceeds the promised execution time. The sporadic tasks are handled by a server policy. The analysis is carried out on this particular model.

Gardner et al. [Gar99, GL99], in their stochastic time demand analysis, introduce worst-case scenarios with respect to task release times in order to compute a lower bound for the probability that a job meets its deadline. Their approach however does not consider data dependencies among tasks and applications implemented on multiprocessors.

Zhou et al. [ZHS99] and Hu et al. [HZS01] root their work in Tia's. However, they do not intend to give per-task guarantees, but

characterise the fitness of the entire task set. Because they consider all possible combinations of execution times of all requests up to a time moment, the analysis can be applied only to small task sets due to complexity reasons.

De Veciana et al. [dJG00] address a different type of problem. Having a task graph and an imposed deadline, their goal is to determine the path that has the highest probability to violate the deadline. In this case, the problem is reduced to a non-linear optimisation problem by using an approximation of the convolution of the probability densities.

A different school of thought [Leh96, Leh97] addresses the problem under special assumptions, such that the system exhibits "heavy traffic", i.e. the processor loads are close to 1. The system is modelled as a continuous state Markov model, in which the state comprises the task laxities, i.e. the time until their deadlines. Under heavy traffic, such a stochastic process converges to a Brownian motion with drift and provides a simple solution. The theory was further extended by Harrison and Nguyen [HN93], Williams [Wil98] and others [PKH01, DLS01, DW93], by modelling the application as a multi-class queueing network and analysing it in heavy traffic.

As far as we are aware, there are two limitations that restrict the applicability of heavy traffic theory to real-time systems. Firstly, heavy traffic theory assumes Poisson task arrival processes and execution times with exponentially distributed probabilities. Secondly, as heavy traffic leads to very long (infinite) queues of ready-to-run tasks, the probability for a job to meet its deadline is almost 0 unless the deadline is very large. Designing a system such that it exhibits heavy traffic is thus undesirable.

Other researchers, such as Kleinberg et al. [KRT00] and Goel and Indyk [GI99], apply approximate solutions to problems exhibiting stochastic behaviour but in the context of load balancing, bin packing and knapsack problems. Moreover, the probability distributions they consider are limited to a few very particular cases.

Díaz et al. [DGK+02] derive the expected deadline miss ratio from the probability distribution function of the response time of a task. The response time is computed based on the system-level backlog at the beginning of each hyperperiod, i.e. the residual execution times of the jobs at those time moments. The stochastic process of the system-level backlog is Markovian and its stationary solution can be computed. Díaz et al. consider only sets of independent tasks and the task execution times may assume values only over discrete sets. In their approach, complexity is mastered by trimming the transition probability matrix of the underlying Markov chain or by deploying iterative methods, both at the expense of result accuracy. According to the published results, the method is exercised only on extremely small task sets.

Kalavade and Moghé [KM98] consider task graphs where the task execution times are arbitrarily distributed over discrete sets. Their analysis is based on Markovian stochastic processes too. Each state in the process is characterised by the executed time and lead-time. The analysis is performed by solving a system of linear equations. Because the execution time is allowed to take only a finite (most likely small) number of values, such a set of equations is small.

Kim and Shin [KS96] consider applications that are implemented on multiprocessors and modelled them as queueing networks. They

restricted the task execution times to exponentially distributed ones, which reduces the complexity of the analysis. The tasks were considered to be scheduled according to a particular policy, namely first-come-first-served (FCFS). The underlying mathematical model is then the appealing continuous time Markov chain.

In the context of multiprocessor systems, our work significantly extends the one by Kim and Shin [KS96]. Thus, we consider arbitrary execution time probability density functions (Kim and Shin consider only exponential ones) and we address a much larger class of scheduling policies (as opposed to FCFS considered by them, or fixed priority scheduling considered by most of the previous work). Moreover, our approach is applicable in the case of arbitrary processor loads as opposed to the heavy traffic school of thought.

Our work is mostly related to the ones of Zhou et al. [ZHS99], Hu et al. [HZS01], Kalavade and Moghé [KM98] and Díaz et al. [DGK+02]. It differs from the others mostly by considering less restricted application classes. As opposed to Kalavade and Moghé's work and to Díaz et al.'s work, we consider *continuous* ETPDFs. In addition to Díaz et al.'s approach, we consider task sets with dependencies among tasks. Also, we accept a much larger class of scheduling policies than the fixed priority ones considered by Zhou and Hu. Moreover, our original way of concurrently constructing and analysing the underlying stochastic process, while keeping only the needed stochastic process states in memory, allows us to consider larger applications.

Chapter 3

System Modelling

This chapter introduces the notations and application model used throughout the thesis. The hardware model presented in this chapter is used throughout Part II.

3.1 Hardware Model

The hardware model consists of a set of *processing elements*. These can be programmable processors of any kind (general purpose, controllers, DSPs, ASIPs, etc.). Let $PE = \{PE_1, PE_2, \ldots, PE_p\}$ denote the set of processing elements. A *bus* may connect two or more processing elements in the set PE. Let $B = \{B_1, B_2, \ldots, B_l\}$ denote the set of buses. Data sent along a bus by a processing element connected to that bus may be read by all processing elements connected to that bus.

Unless explicitly stated, the two types of hardware resources, processing elements and buses, will not be differently treated in the scope of this part of the thesis, and therefore they will be denoted with the general term of *processors*. Let $M = p + l$ denote the number of processors and let $P = PE \cup B = \{P_1, P_2, \ldots, P_M\}$ be the set of processors.

Figure 3.1 depicts a hardware platform consisting of three processing elements and two buses. Bus B_1 connects all processing elements, while bus B_2 is a point-to-point link connecting processing element PE_1 and processing element PE_2.

3.2 Application Model

3.2.1 Functionality

The functionality of an application is modelled as a set of *processing tasks*, denoted with $t_1, t_2, \ldots, t_n$. A processing task is a piece of work

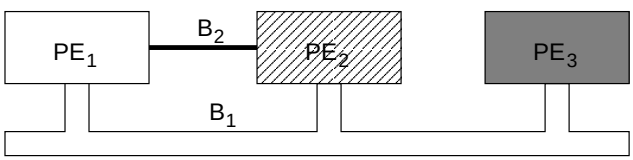


Figure 3.1: Hardware model

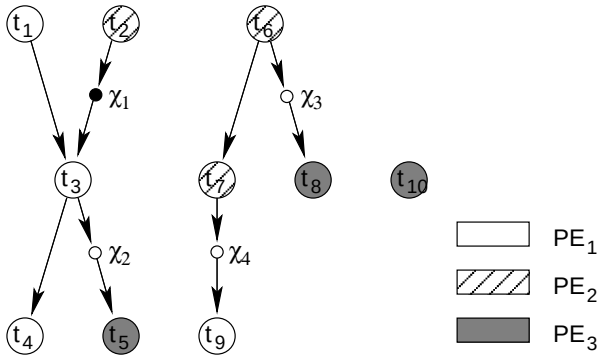


Figure 3.2: Application model

that has a conceptual unity and is assigned to a processing element. Examples of processing tasks are performing a discrete cosine transform on a stream of data in a video decoding application or the encryption of a stream of data in the baseband processing of a mobile communication application. Let PT denote the set of processing tasks. Processing tasks are graphically represented as large circles, as shown in Figure 3.2.

Processing tasks may pass messages to each other. The passing of a message is modelled as a *communication task*, denoted with χ. Let CT denote the set of communication tasks. They are graphically depicted as small circles, as shown in Figure 3.2.

Unless explicitly stated, the processing and the communication tasks will not be differently treated in the scope of Part II of the thesis, and therefore they will be denoted with the general term of *tasks*. Let N be the number of tasks and $T = PT \cup CT = \{\tau_1, \tau_2, \ldots, \tau_N\}$ denote the set of tasks.

The passing of a message between tasks τ_i and τ_j enforces *data dependencies* between the two tasks. Data dependencies are graphically depicted as arrows from the sender task to the receiver task, as shown in Figure 3.2.

The task that sends the message is the *predecessor* of the receiving task, while the receiving task is the *successor* of the sender. The set of predecessors of task τ is denoted with ${}^\circ\tau$, while the set of successors of task τ with τ°. A communication task has exactly one predecessor and one successor and both are processing tasks. For illustration (Figure 3.2), task t_3 has two predecessors, namely the processing task t_1 and the communication task χ_1, and it has two successors, namely tasks t_4 and χ_2.

Tasks with no predecessors are called *root* tasks, while tasks with no successors are called *leaf* tasks. In Figure 3.2 tasks t_1, t_2, t_6, and t_{10} are root tasks, while tasks t_4, t_5, t_8, t_9, and t_{10} are leaf tasks.

Let us consider a sequence of tasks $(\tau_1, \tau_2, \ldots, \tau_k)$, $k > 1$. If there exists a data dependency between tasks τ_i and τ_{i+1}, $\forall 1 \leq i < k$, then the sequence $(\tau_1, \tau_2, \ldots, \tau_k)$ forms a *computation path* of length k. We say that the computation path leads from task τ_1 to task τ_k. Task τ_i is an *ancestor task* of task τ_j if there exists a computation path from task τ_i to task τ_j. Complementarily, we say that task τ_i is a *descendant task* of task τ_j if there exists a computation path from task τ_j to task τ_i. We do not allow circular dependencies, i.e. no task can be both the ancestor

and the descendant of another task. In Figure 3.2, $(t_2, \chi_1, t_3, \chi_2, t_5)$ is an example of a computation path of length 5, and task χ_1 is an ancestor of tasks t_3, t_4, t_5, and χ_2.

We define the relation $\gamma \subset T \times T$ as follows:

- $(\tau, \tau) \in \gamma, \forall \tau \in T$,
- $(\tau_i, \tau_j) \in \gamma, \forall \tau_i, \tau_j \in T, \tau_i \neq \tau_j$ iff
  - they have at least one common ancestor, or
  - they have at least one common successor, or
  - they are in a predecessor-successor relationship.

As γ is a reflexive, symmetric, and transitive relation, it is an equivalence relation. Hence, it partitions the set of tasks T into g subsets, denoted with V_i, $1 \leq i \leq g$ ($\cup_{i=1}^{g} V_i = T \wedge V_i \cap V_j = \varnothing, \forall 1 \leq i, j \leq g, i \neq j$). Thus, an application consists of a set $\Gamma = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_g\}$ of g ***task graphs***, $\Gamma_i = (V_i, E_i \subset V_i \times V_i)$, $1 \leq i \leq g$. A directed edge $(\tau_a, \tau_b) \in E_i$, $\tau_a, \tau_b \in V_i$, represents the data dependency between tasks τ_a and τ_b, denoted $\tau_a \rightarrow \tau_b$.

The application example in Figure 3.2 consists of three task graphs: $\Gamma_1 = (\{t_1, t_2, t_3, t_4, t_5, \chi_1, \chi_2\}, \{(t_1, t_3), (t_2, \chi_1), (\chi_1, t_3), (t_3, t_4), (t_3, \chi_2), (\chi_2, t_5)\})$, $\Gamma_2 = (\{t_6, t_7, t_8, t_9, \chi_3, \chi_4\}, \{(t_6, t_7), (t_7, \chi_4), (\chi_4, t_9), (t_6, \chi_3), (\chi_3, t_8)\})$, and $\Gamma_3 = (\{t_{10}\}, \varnothing)$.

3.2.2 Periodic Task Model

Task *instantiations* (also known as *jobs*) *arrive* periodically. The i^{th} job of task τ is denoted (τ, i), $i \in \mathbb{N}$.

Let $\Pi_T = \{\pi_i \in \mathbb{N} : \tau_i \in T\}$ denote the set of *task periods*, or job inter-arrival times, where π_i is the period of task τ_i. Instantiation $u \in \mathbb{N}$ of task τ_i demands execution (the job is *released* or the job *arrives*) at time moment $u \cdot \pi_i$. The period π_i of any task τ_i is assumed to be a common multiple of all periods of its predecessor tasks (π_j divides π_i, where $\tau_j \in {}^\circ\tau_i$). Let k_{ij} denote $\frac{\pi_i}{\pi_j}$, $\tau_j \in {}^\circ\tau_i$. Instantiation $u \in \mathbb{N}$ of task τ_i may start executing only if instantiations $u \cdot k_{ij}, u \cdot k_{ij} + 1, \ldots, u \cdot k_{ij} + k_{ij} - 1$ of tasks τ_j, $\forall \tau_j \in {}^\circ\tau_i$, have completed their execution.

Let $\Pi_\Gamma = \{\pi_{\Gamma_1}, \pi_{\Gamma_2}, \ldots, \pi_{\Gamma_g}\}$ denote the set of *task graph periods* where π_{Γ_j} denotes the period of the task graph Γ_j. π_{Γ_j} is equal to the least common multiple of all π_i, where π_i is the period of τ_i and $\tau_i \in V_j$. Task $\tau_i \in V_j$ is instantiated $J_i = \frac{\pi_{\Gamma_j}}{\pi_i}$ times during one instantiation of task graph Γ_j. The k^{th} instantiation of task graph Γ_j, $k \geq 0$, denoted (Γ_j, k), is composed of the jobs (τ_i, u), where $\tau_i \in V_j$ and $u \in \{k \cdot J_i, k \cdot J_i + 1, \ldots, k \cdot J_i + J_i - 1\}$. In this case, we say that task instantiation (τ_i, u) belongs to task graph instantiation (Γ_j, k) and we denote it with $(\tau_i, u) \in (\Gamma_j, k)$.

The model, where task periods are integer multiples of the periods of predecessor tasks, is more general than the model assuming equal task periods for tasks in the same task graph. This is appropriate, for instance, when modelling protocol stacks. For example, let us consider a part of baseband processing on the GSM radio interface [MP92]. A data frame is assembled out of 4 radio bursts. One task implements the decoding of radio bursts. Each time a burst is decoded, the result is sent to the frame assembling task. Once the frame assembling task gets all the needed data, that is every 4 invocations of the burst decoding task, the frame assembling task is invoked. This way of modelling is more

modular and natural than a model assuming equal task periods, which would have crammed the four invocations of the radio burst decoding task in one task. We think that more relaxed models than ours, with regard to relations between task periods, are not necessary, as such applications would be more costly to implement and are unlikely to appear in common engineering practice.

3.2.3 Mapping

Processing tasks are *mapped* on processing elements and communication tasks are mapped on buses. All instances of a processing task are executed by the same processing element on which the processing task is mapped. Analogously, all instances of a message are conveyed by the bus on which the corresponding communication task is mapped.

Let $MapP : PT \rightarrow PE$ be a surjective function that maps processing tasks on the processing elements. $MapP(t_i) = P_j$ indicates that processing task t_i is executed on the processing element P_j. Let $MapC : CT \rightarrow B$ be a surjective function that maps communication tasks on buses. $MapC(\chi_i) = B_j$ indicates that the communication task χ_i is performed on the bus B_j. For notation simplicity, $Map : T \rightarrow P$ is defined, where $Map(\tau_i) = MapP(\tau_i)$ if $\tau_i \in PT$ and $Map(\tau_i) = MapC(\tau_i)$ if $\tau_i \in CT$. Conversely, let $\mathcal{T}_p = \{\tau \in T : Map(\tau) = p \in P\}$ denote the set of tasks that are mapped on processor p. Let $\mathcal{T}_\tau$ be a shorthand notation for $\mathcal{T}_{Map(\tau)}$.

The mapping is graphically indicated by the shading of the task. In Figure 3.2, tasks t_1, t_3, t_4, and t_9 are mapped on processing element PE_1, tasks t_2, t_6, and t_7 on processing element PE_2, and tasks t_5, t_9, and t_{10} on processing element PE_3. Communication task χ_1 is mapped on bus B_2 and communication tasks χ_2, χ_3, and χ_4 on bus B_1. The corresponding system architecture is shown in Figure 3.1.

3.2.4 Execution Times

For a processing task t_i, $\forall 1 \leq i \leq n$, let Ex_{t_i} denote its execution time on processing element $MapP(t_i)$. Let ϵ_{t_i} be the probability density of Ex_{t_i}.

First, we discuss the modelling of the communication time between two processing tasks that are mapped on the same processing element. Let t_i and t_j be any two processing tasks such that task t_i is a predecessor of task t_j ($t_i \in {}^\circ t_j$) and tasks t_i and t_j are mapped on the same processing element ($MapP(t_i) = MapP(t_j)$). In this case, the time of the communication between task t_i and t_j is considered to be part of the execution time of task t_i. Thus, the execution time probability density ϵ_{t_i} accounts for this intra-processor communication time.

Next, we discuss the modelling of the communication time between two processing tasks that are mapped on different processing elements. Let t_i and t_j be two processing tasks, let χ be a communication task, let PE_a and PE_b be two distinct processing elements and let B be a bus such that all of the following statements are true:

- Processing tasks t_i and t_j are mapped on processing elements PE_a and PE_b respectively ($MapP(t_i) = PE_a$ and $MapP(t_j) = PE_b$).
- Communication task χ is mapped on bus B ($MapC(\chi) = B$).

- Bus B connects processing elements PE_a and PE_b.
- Task χ is a successor of task t_i and a predecessor of task t_j ($\chi \in t_i^\circ \wedge \chi \in {}^\circ t_j$).

The transmission time of the message that is passed between tasks t_i and t_j on the bus B is modelled by the execution time Ex_χ of the communication task χ. Let ϵ_χ denote the probability density of Ex_χ.

Without making any distinction between processing and communication tasks, we let Ex_i denote an execution (communication) time of an instantiation of task $\tau_i \in T$ and we let $ET = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_N\}$ denote the set of N *execution time probability density functions (ETPDFs)*.

3.2.5 Real-Time Requirements

The real-time requirements are expressed in terms of deadlines. Let $\Delta_T = \{\delta_i \in \mathbb{N} : \tau_i \in T\}$ denote the set of *task deadlines*. δ_i is the deadline of task τ_i. If job (τ_i, u) has not completed its execution at time $u \cdot \pi_i + \delta_i$, then the job is said to have missed its deadline.

Let $\Delta_\Gamma = \{\delta_{\Gamma_j} \in \mathbb{N} : 1 \leq j \leq g\}$ denote the set of task graph deadlines, where δ_{Γ_j} is the deadline of task graph Γ_j. If there exists at least one task instantiation $(\tau_i, u) \in (\Gamma_j, k)$, such that (τ_i, u) has not completed its execution at time moment $k \cdot \pi_{\Gamma_j} + \delta_{\Gamma_j}$, we say that task graph instantiation (Γ_j, k) has missed its deadline.

If $D_i(t)$ denotes the number of jobs of task τ_i that have missed their deadline over a time span t and $A_i(t) = \lfloor \frac{t}{\pi_i} \rfloor$ denotes the total number of jobs of task τ_i over the same time span, then $\lim_{t\to\infty} \frac{D_i(t)}{A_i(t)}$ denotes the *expected deadline miss ratio* of task τ_i. Similarly, we define the expected deadline miss ratio of task graph Γ_j as the long-term ratio between the number of instantiations of task graph Γ_j that have missed their deadlines and the total number of instantiations of task graph Γ_j.

Let $Missed_T = \{m_{\tau_1}, m_{\tau_2}, \ldots, m_{\tau_N}\}$ be the set of expected deadline miss ratios per task. Similarly, the set $Missed_\Gamma = \{m_{\Gamma_1}, m_{\Gamma_2}, \ldots, m_{\Gamma_g}\}$ is defined as the set of expected deadline miss ratios per task graph.

The designer may specify upper bounds for tolerated deadline miss ratios, both for tasks and for task graphs. Let $\Theta_T = \{\theta_{\tau_1}, \theta_{\tau_2}, \ldots, \theta_{\tau_N}\}$ be the set of deadline miss thresholds for tasks and let $\Theta_\Gamma = \{\theta_{\Gamma_1}, \theta_{\Gamma_2}, \ldots, \theta_{\Gamma_g}\}$ be the set of deadline miss thresholds for task graphs.

Some tasks or task graphs may be designated as being *critical* by the designer, which means that deadline miss thresholds are not allowed to be violated. The deadline miss *deviation* of task τ, denoted dev_τ, is defined as

$$dev_\tau = \begin{cases} \infty & m_\tau > \theta_\tau, \tau \text{ critical} \\ m_\tau - \theta_\tau & m_\tau > \theta_\tau, \tau \text{ not critical} \\ 0 & m_\tau \leq \theta_\tau. \end{cases} \tag{3.1}$$

Analogously, we define the deadline miss deviation of a task graph.

3.2.6 Late Task Policy

We say that a task graph instantiation (Γ, k), $k \geq 0$, is active in the system at time t if there exists at least one task instantiation $(\tau, u) \in (\Gamma, k)$ such that job (τ, u) has not completed its execution at time t. Let

$Inst_i(t)$ denote the number of active instantiations of task graph Γ_i, $1 \leq i \leq g$, at time t.

For each task graph $\Gamma_i, \forall 1 \leq i \leq g$, we let the designer specify $b_i \in \mathbb{N}^+$, the maximum number of simultaneously active instantiations of task graph Γ_i. Let $Bounds = \{b_i \in \mathbb{N}^+ : 1 \leq i \leq g\}$ be their set.

We consider two different policies for ensuring that no more than b_i instantiations of task graph Γ_i, $\forall 1 \leq i \leq g$, are active in the system at the same time. We call these policies the *discarding* and the *rejection* policy.

We assume that a system applies the same policy (either discarding or rejection) to all task graphs, although our work can be easily extended in order to accommodate task graph specific late task policies.

The Discarding Policy

The discarding policy specifies that whenever a new instantiation of task graph Γ_i, $\forall 1 \leq i \leq g$, arrives and b_i instantiations are already active in the system at the time of the arrival of the new instantiation, the oldest active instantiation of task graph Γ_i is *discarded*. We mean by "oldest" instantiation the instantiation whose arrival time is the minimum among the arrival times of the active instantiations of the same task graph. "Discarding" a task graph implies:

- The running jobs belonging to the task graph to be discarded are immediately removed from the processors they run onto. These jobs are eliminated from the system, i.e. their execution is never resumed and all resources that they occupy (locks, memory, process control blocks, file control blocks, etc.) are freed.
- The ready-to-run and blocked-on-I/O jobs belonging to the task graph to be discarded are immediately removed from the ready-to-run and waiting-on-I/O queues of the scheduler. They are also eliminated from the system.

The Rejection Policy

The rejection policy specifies that whenever a new instantiation of task graph Γ_i, $\forall 1 \leq i \leq g$, arrives and b_i instantiations are active in the system at the time of the arrival of the new instantiation, the new instantiation is not accepted in the system. Thus, all execution requests by jobs belonging to the new instantiation are ignored by the system.

3.2.7 Scheduling Policy

In the common case of more than one task mapped on the same processor, the designer has to decide on a *scheduling policy*. Such a scheduling policy has to be able to unambiguously determine the running task at any time on that processor. The selection of the next task to run is made by a run-time scheduler based on the priority associated to the task. Priorities may be static (the priority of a task does not change in time) or dynamic (the priority of a task changes in time).

We limit the set of accepted scheduling policies to those where the sorting of tasks according to their priority is unique during those time intervals in which the queue of ready tasks is unmodified. For practical purposes, this is not a limitation, as all practically used priority-based scheduling policies [LL73, But97, Fid98, ABD$^+$95], both with

static priority assignment (rate monotonic, deadline monotonic) and with dynamic assignment (earlier deadline first (EDF)), fulfil this requirement.

The scheduling policy is nevertheless restricted to non-preemptive scheduling. This limitation is briefly discussed in Section 4.2.1.

Each processing element or bus may have a different scheduling policy associated to it.

Based on these assumptions, and considering the communication tasks, we are able to model any priority based bus arbitration protocol as, for instance, CAN [Bos91]. [1]

3.3 Illustrative Example

This section illustrates the behaviour of the application shown in Figure 3.2 by means of a Gantt diagram. Tasks are mapped as indicated in Section 3.2.3.

Tasks t_1, t_2, t_3, t_5, χ_1, and χ_2 have a period of 6 and task t_4 has a period of 12. Consequently, task graph Γ_1 has a period $\pi_{\Gamma_1} = 12$. Tasks t_6, t_7, t_8, χ_3, and χ_4 have a period of 4 and task τ_9 has a period of 8. Thus, $\pi_{\Gamma_2} = 8$. Task t_{10} has a period of 3. For all tasks and task graphs of this example, their deadline is equal to their period ($\delta_i = \pi_i$, $1 \leq i \leq 14$ and $\delta_{\Gamma_i} = \pi_{\Gamma_i}$, $1 \leq i \leq 3$).

The late task policy for this example is the discarding policy. The set of bounds on the number of simultaneously active instantiations of the same task graph is $Bounds = \{1, 1, 2\}$.

The deployed scheduling policy is fixed priority for this example. As the task priorities do not change, this policy obviously satisfies the restriction that the sorting of tasks according to their priorities must be invariable during the intervals in which the queue of ready tasks does not change. Task t_7 has a higher priority than task t_6, which in turn has a higher priority than task t_2. Task t_{10} has a higher priority than task t_8, which in turn has a higher priority than task t_5. Task t_9 has a higher priority than any of the tasks t_1, t_3, and t_4. Message χ_4 has a higher priority than message χ_3, which in turn has a higher priority than message χ_2.

A Gantt diagram illustrating a possible task execution over a span of 20 time units is depicted in Figure 3.3. The Ox axes corresponds to time, while each processor, shown on the Oy axis, has an associated Ox axis. The job executions are depicted as rectangles stretching from the point on the Ox axis that corresponds to the start time of the job execution to the point on the Ox axis that corresponds to its finishing time. The different task graphs are depicted in different shades in this figure. Vertical lines of different line patterns are used for better readability.

Job 2 of task t_{10} arrives at time 6 and is ready to run. However, processing element PE_3 is busy executing task t_8. Therefore, job $(t_{10}, 2)$

[1] Time division multiple access bus protocols, such as the TTP [TTT99], could be modelled using dynamic priorities. For example, all communication tasks that are not allowed to transmit on a bus at a certain moment of time have priority $-\infty$. However, in this case, the sorting of tasks according to their priorities is not any more unique between two consecutive events that change the set of ready tasks. The reason is that between two such events, a time slot may arrive when a certain communication is allowed and, thus, the priorities of communication tasks are changed. Therefore, time-triggered communication protocols are not supported by our analysis method.

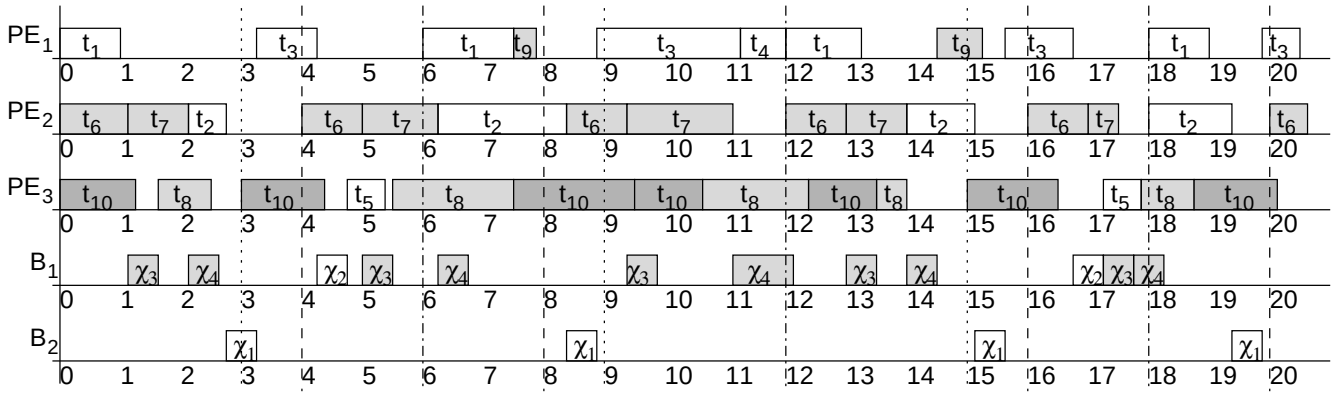


Figure 3.3: Gantt diagram

starts its execution later, at time7.5. The execution of the job finishes at time 9.5. At time 9, job 3 of task t_{10} arrives and is ready to run. Thus, we observe that between times 9 and 9.5 two instantiations of task graph Γ_3 are active. This is allowed as $b_2 = 2$.

Instantiation 0 of task graph Γ_1 illustrates a discarding situation. Job 1 of task t_3, which arrived at time 6, starts its execution at time 8.875, when communication task χ_1 has finished, and finishes at time 11. At this time, the message between jobs $(t_3, 1)$ and $(t_5, 1)$ is ready to be sent on bus B_1 (an instance of communication task χ_2 is "ready-to-run"). Nevertheless, the message cannot be sent at time 11 as the bus is occupied by an instance of communication task χ_4. Message χ_4 is still being sent at the time 12, when a new instantiation of task graph Γ_1 arrives. At time 12, the following jobs belonging to instantiation 0 of task graph Γ_1 have not yet completed their execution: $(t_4, 0)$, $(t_5, 1)$, and $(\chi_2, 1)$. They are in the states "running", "waiting on I/O", and "ready-to-run" respectively. Because at most one instantiation of task graph Γ_1 is allowed to be active at any time, instantiation 0 must be discarded at time 12. Hence, job $(t_4, 0)$ is removed from processing element PE_1, job $(t_5, 1)$ is removed from the waiting-on-I/O queue of the processing element PE_2, and job $(\chi_2, 1)$ is removed from the ready-to-run queue of bus B_1.

The deadline miss ratio of Γ_3 over the interval $[0, 18)$ is $1/6$, because there are 6 instantiations of task graph Γ_3 in this interval and one of them, instantiation 2, which arrived at time 6, missed its deadline. When analysing this system, the *expected* deadline miss ratio of Γ_3 (the ratio of the number instantiations that missed their deadline and the total number of instantiations over an infinite time interval) is 0.08. The expected deadline miss ratios of Γ_1 and Γ_2 are 0.4 and 0.15 respectively.

Chapter 4

Analysis of Monoprocessor Systems

This chapter presents an exact approach for analytically determining the expected deadline miss ratios of task graphs with stochastic task execution times in the case of monoprocessor systems.

First, we give the problem formulation (Section 4.1). Second, we present the analysis procedure based on an example before we give the precise algorithm (Section 4.2). Third, we evaluate the efficiency of the analysis procedure by means of experiments (Section 4.3). Section 4.4 presents some extensions of the assumptions. Last, we discuss the limitations of the approach presented in this chapter and we hint on the possible ways to overcome them.

4.1 Problem Formulation

The formulation of the problem to be solved in this chapter is the following:

4.1.1 Input

The input of the analysis problem to be solved in this chapter is given as follows:

- The set of task graphs Γ,
- The set of task periods Π_T and the set of task graph periods Π_Γ,
- The set of task deadlines Δ_T and the set of task graph deadlines Δ_Γ,
- The set of execution time probability density functions ET,
- The late task policy is the discarding policy,
- The set $Bounds = \{b_i \in \mathbb{N}\backslash\{0\} : 1 \leq i \leq g\}$, where b_i is the maximum numbers of simultaneously active instantiations of task graph Γ_i, and
- The scheduling policy.

4.1.2 Output

The result of the analysis is the set $Missed_T$ of expected deadline miss ratios for each task and the set $Missed_\Gamma$ of expected deadline miss ratios for each task graph.

4.1.3 Limitations

We assume the discarding late task policy. A discussion on discarding versus rejection policy is presented in Section 4.3.5.

4.2 Analysis Algorithm

The goal of the analysis is to obtain the expected deadline miss ratios of the tasks and task graphs. These can be derived from the behaviour of the system. The behaviour is defined as the evolution of the system through a *state space* in time. A *state* of the system is given by the values of a set of variables that characterise the system. Such variables may be the currently running task, the set of ready tasks, the current time and the start time of the current task..

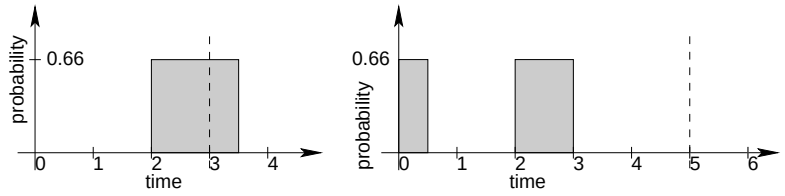
Due to the considered periodic task model, the task arrival times are deterministically known. However, because of the stochastic task execution times, the completion times and implicitly the running task at an arbitrary time instant or the state of the system at that instant cannot be deterministically predicted. The mathematical abstraction best suited to describe and analyse such a system with random character is the stochastic process.

In this section, we first sketch the stochastic process construction and analysis procedure based on a simplified example. Then the memory efficient construction of the stochastic process underlying the application is detailed. Third, the algorithm is refined in order to handle multiple concurrently active instantiations of the same task graph. Finally, the complete algorithm is presented.

4.2.1 The Underlying Stochastic Process

Let us define LCM as the least common multiple of the task periods. For simplicity of the exposition, we first assume that at most one instantiation of each task graph is tolerated in the system at the same time ($b_i = 1, \forall 1 \leq i \leq g$). In this case, the set of time moments when all late tasks are discarded include the sequence $LCM, 2 \cdot LCM, \ldots, k \cdot LCM, \ldots$ because at these moments new instantiations of all tasks arrive. The system behaves at these time moments as if it has just been started. The time moments $k \cdot LCM$, $k \in \mathbb{N}$ are called regeneration points. Regardless of the chosen definition of the state space of the system, the system states at the renewal points are equivalent to the initial state, which is unique and deterministically known. Thus, the behaviour of the system over the intervals $[k \cdot LCM, (k+1) \cdot LCM)$, $k \in \mathbb{N}$, is statistically equivalent to the behaviour over the time interval $[0, LCM)$. Therefore, in the case when $b_i = 1$, $1 \leq i \leq g$, it is sufficient to analyse the system solely over the time interval $[0, LCM)$.

One could choose the following state space definition: $S = \{(\tau, W, t) : \tau \in T, W \in$ set of all multisets of $T, t \in \mathbb{R}\}$, where τ represents the



(a) ϵ_1 (b) ϵ_2

Figure 4.1: ETPDFs of tasks τ_1 (ϵ_1) and τ_2 (ϵ_2)

currently running task, W stands for the multiset[1] of ready tasks at the start time of the running task, and t represents the start time of the currently running task. A state change occurs at the time moments when the scheduler has to decide on the next task to run. This happens

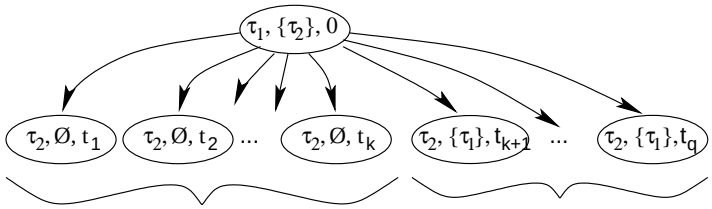
- when a task completes its execution, or
- when a task arrives and the processor is idle, or
- when the running task graph has to be discarded.

The point we would like to make is that, by choosing this state space, the information provided by a state $s_i = (\tau_i, W_i, t_i)$, together with the current time, is sufficient to determine the next system state $s_j = (\tau_j, W_j, t_j)$. The time moment when the system entered state s_i, namely t_i, is included in s_i. Because of the deterministic arrival times of tasks, based on the time moments t_j and on t_i, we can derive the multiset of tasks that arrived in the interval $(t_i, t_j]$. The multiset of ready tasks at time moment t_i, namely W_i, is also known. We also know that τ_i is not preempted between t_i and t_j. Therefore, the multiset of ready tasks at time moment t_j, prior to choosing the new task to run, is the union of W_i and the tasks arrived during the interval $(t_i, t_j]$. Based on this multiset and on the time t_j, the scheduler is able to predictably choose the new task to run. Hence, in general, knowing a current state s and the time moment t when a transition out of state s occurs, the next state s' is unambiguously determined.

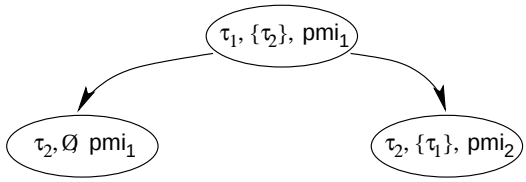
The following example is used throughout this subsection in order to discuss the construction of the stochastic process. The system consists of one processor and the following application: $\Gamma = \{(\{\tau_1\}, \varnothing), (\{\tau_2\}, \varnothing)\}$, $\Pi = \{3, 5\}$, i.e. a set of two independent tasks with corresponding periods 3 and 5. The tasks are scheduled according to a non-preemptive EDF scheduling policy [LL73]. LCM, the least common multiple of the task periods is 15. For simplicity, in this example it is assumed that the relative deadlines equal the corresponding periods ($\delta_i = \pi_i$). The ETPDFs of the two tasks are depicted in Figure 4.1. Note that ϵ_1 contains execution times larger than the deadline δ_1.

Let us assume a state representation like the one introduced above: each process state contains the identity of the currently running task, its start time and the multiset of ready task at the start time of the currently running one. For our application example, the initial state is $(\tau_1, \{\tau_2\}, 0)$, i.e. task τ_1 is running, it has started to run at time moment

[1] If $b_i = 1, \forall 1 \leq i \leq g$, then W is a set.



(a) Individual task completion times



(b) Intervals containing task completion times

Figure 4.2: State encoding

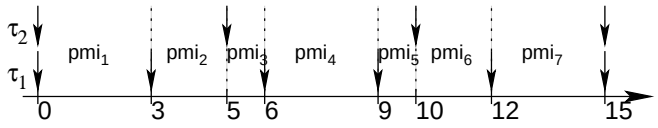


Figure 4.3: Priority monotonicity intervals

0 and task τ_2 is ready to run, as shown in Figure 4.2(a). $t_1, t_2, \ldots, t_q$ in the figure are possible finishing times for the task τ_1 and, implicitly, possible starting times of the waiting instantiation of task τ_2. The number of next states equals the number of possible execution times of the running task in the current state. In general, because the ETPDFs are continuous, the set of state transition moments form a dense set in $\mathbb{R}$ leading to an underlying stochastic process theoretically of uncountable state space. In practice, the stochastic process is extremely large, depending on the discretisation resolution of the ETPDFs. Even in the case when the task execution time probabilities are distributed over a discrete set, the resulting underlying process becomes prohibitively large and practically impossible to solve.

In order to avoid the explosion of the underlying stochastic process, in our approach, we have grouped time moments into equivalence classes and, by doing so, we limited the process size explosion. Thus, practically, a set of equivalent states is represented as a single state in the stochastic process.

As a first step to the analysis, the interval $[0, LCM)$ is partitioned in disjunct intervals, the so-called *priority monotonicity intervals (PMI)*. The concept of PMI (called in their paper "state") was introduced by Zhou et al. [ZHS99] in a different context, unrelated to the construction of a stochastic process. A PMI is delimited by task arrival times and task execution deadlines. Figure 4.3 depicts the PMIs for the example above. The only restriction imposed on the scheduling policies accepted

by our approach is that inside a PMI the ordering of tasks according to their priorities is not allowed to change. This allows the scheduler to predictably choose the next task to run, regardless of the completion time within a PMI of the previously running task. As mentioned in Section 3.2.7, all the widely used scheduling policies we are aware of (rate monotonic (RM), EDF, first come first served (FCFS), LLF, etc.) exhibit this property, as mentioned before.

Consider a state s characterised by (τ_i, W, t): τ_i is the currently running task, it has been started at time t, and W is the multiset of ready tasks. Let us consider two next states derived from s: s_1 characterised by (τ_j, W_1, t_1) and s_2 by (τ_k, W_2, t_2). Let t_1 and t_2 belong to the same PMI. This means that no task instantiation has arrived or been discarded in the time interval between t_1 and t_2, and the relative priorities of the tasks inside the set W have not changed between t_1 and t_2. Thus, $\tau_j = \tau_k =$ the highest priority task in the multiset W, and $W_1 = W_2 = W \backslash \{\tau_j\}$. It follows that all states derived from state s that have their time t belonging to the same PMI have an identical currently running task and identical sets of ready tasks. Therefore, instead of considering individual times we consider time intervals, and we group together those states that have their associated start time inside the same PMI. With such a representation, the number of next states of a state s equals the number of PMIs the possible execution time of the task that runs in state s is spanning over.

We propose a representation in which a stochastic process state is a triplet (τ, W, pmi), where τ is the running task, W the multiset of ready tasks at the start time of task τ, and pmi is the PMI containing the start time of the running task. In our example, the execution time of task τ_1 (which is in the interval $[2, 3.5]$, as shown in Figure 4.1(a)) is spanning over the PMIs pmi_1— $[0, 3)$—and pmi_2—$[3, 5)$. Thus, there are only two possible states emerging from the initial state, as shown in Figure 4.2(b).

Figure 4.4 depicts a part of the stochastic process constructed for our example. The initial state is $s_1 : (\tau_1, \{\tau_2\}, pmi_1)$. The first field indicates that an instantiation of task τ_1 is running. The second field indicates that an instantiation of task τ_2 is ready to execute. The third field shows the current PMI (pmi_1—$[0, 3)$). If the instantiation of task τ_1 does not complete until time moment 3, then it will be discarded. The state s_1 has two possible next states. The first one is state $s_2 : (\tau_2, \varnothing, pmi_1)$ and corresponds to the case when the τ_1 completes before time moment 3. The second one is state $s_3 : (\tau_2, \{\tau_1\}, pmi_2)$ and corresponds to the case when τ_1 was discarded at time moment 3. State s_2 indicates that an instantiation of task τ_2 is running (it is the instance that was waiting in state s_1), that the PMI is pmi_1—$[0, 3)$—and that no task is waiting. Consider state s_2 to be the new current state. Then the next states could be state $s_4 : (-, \varnothing, pmi_1)$ (task τ_2 completes before time moment 3 and the processor is idle), state $s_5 : (\tau_1, \varnothing, pmi_2)$ (task τ_2 completes at a time moment sometime between 3 and 5), or state $s_6 : (\tau_1, \{\tau_2\}, pmi_3)$ (the execution of task τ_2 reaches over time moment 5 and, hence, it is discarded at time moment 5). The construction procedure continues until all possible states corresponding to the time interval $[0, LCM)$, i.e. $[0, 15)$, have been visited.

Let $\mathcal{P}_i$ denote the set of predecessor states of a state s_i, i.e. the set of all states that have s_i as a next state. The set of successor states of a state s_i consists of those states that can directly be reached from

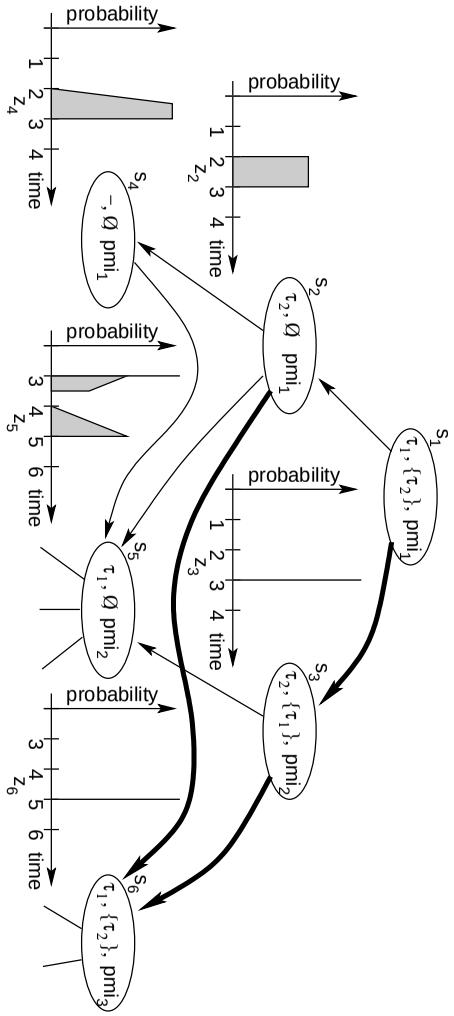


Figure 4.4: Stochastic process example

state s_i. Let Z_i denote the time when state s_i is entered. State s_i can be reached from any of its predecessor states $s_j \in \mathcal{P}_i$. Therefore, the probability $\mathsf{P}(Z_i \leq t)$ that state s_i is entered before time t is a weighted sum over j of probabilities that the transitions $s_j \rightarrow s_i$, $s_j \in \mathcal{P}_i$, occur before time t. The weights are equal to the probability $\mathsf{P}(s_j)$ that the system is in state s_j prior to the transition. Formally, $\mathsf{P}(Z_i \leq t) = \sum_{j \in \mathcal{P}_i} \mathsf{P}(Z_{ji} \leq t | s_j) \cdot \mathsf{P}(s_j)$, where Z_{ji} is the time of transition $s_j \rightarrow s_i$. Let us focus on Z_{ji}, the time of transition $s_j \rightarrow s_i$. If the state transition occurs because the processor is idle and a new task arrives or because the running task graph has to be discarded, the time of the transition is deterministically known as task arrivals and deadlines have fixed times. If, however, the cause of the state transition is a task completion, the time Z_{ji} is equal to to $Z_j + Ex_\tau$, where task τ is the task that runs in state s_j and whose completion triggers the state transition. Because Z_{ji} is a sum involving the random variable Ex_τ, Z_{ji} too is a random variable. Its probability density function, is computed as the convolution $z_j * \epsilon_\tau = \int_0^\infty z_j(t-x) \cdot \epsilon_\tau(x) dx$ of the probability density functions of the terms.

Let us illustrate the above, based on the example depicted in Figure 4.4. z_2, z_3, z_4, z_5, and z_6 are the probability density functions of Z_2, Z_3, Z_4, Z_5, and Z_6 respectively. They are shown in Figure 4.4 to the left of their corresponding states $s_2, s_3, \ldots, s_6$. The transition from state s_4 to state s_5 occurs at a precisely known time instant, time 3, at which a new instantiation of task τ_1 arrives. Therefore, z_5 will contain a scaled Dirac impulse at the beginning of the corresponding PMI. The scaling coefficient equals the probability of being in state s_4 (the integral of z_4, i.e. the shaded surface below the z_4 curve). The probability density function z_5 results from the superposition of $z_2 * \epsilon_2$ (because task τ_2 runs in state s_2) with $z_3 * \epsilon_2$ (because task τ_2 runs in state s_3 too) and with the aforementioned scaled Dirac impulse over pmi_2, i.e. over the time interval $[3, 5)$.

The probability of a task missing its deadline is easily computed from the transition probabilities of those transitions that correspond to a deadline miss of a task instantiation (the thick arrows in Figure 4.4, in our case). The probabilities of the transitions out of a state s_i are computed exclusively from the information stored in that state s_i. For example, let us consider the transition $s_2 \rightarrow s_6$. The system enters state s_2 at a time whose probability density is given by z_2. The system takes the transition $s_2 \rightarrow s_6$ when the attempted completion time of τ_2 (running in s_2) exceeds 5. The completion time is the sum of the starting time of τ_2 (whose probability density is given by z_2) and the execution time of τ_2 (whose probability density is given by ϵ_2). Hence, the probability density of the completion time of τ_2 is given by the convolution $z_2 * \epsilon_2$ of the above mentioned densities. Once this density is computed, the probability of the completion time being larger than 5 is easily computed by integrating the result of the convolution over the interval $(5, \infty)$. If τ_2 in s_2 completes its execution at some time $t \in [3, 5)$, then the state transition $s_2 \rightarrow s_5$ occurs (see Figure 4.4). The probability of this transition is computed by integrating $z_2 * \epsilon_2$ over the interval $[3, 5)$.

As can be seen, by using the PMI approach, some process states have more than one incident arc, thus keeping the graph "narrow". This is because, as mentioned, one process state in our representation

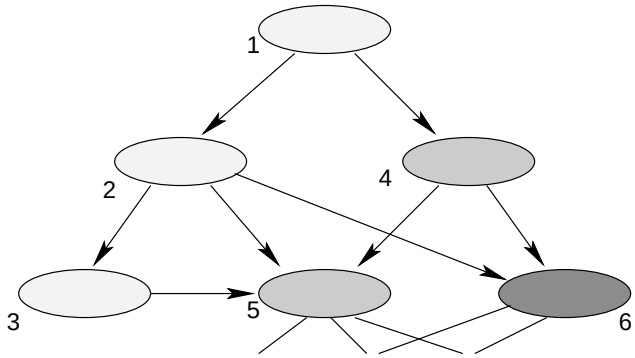


Figure 4.5: State selection order

captures several possible states of a representation considering individual times (see Figure 4.2(a)).

The non-preemption limitation could, in principle, be overcome if we extended the information stored in the state of the underlying stochastic process. Namely, the *residual* run time probability distribution function of a task instantiation, i.e. the PDF of the time a preempted instantiation still has to run, has to be stored in the stochastic process state. This would several times multiply the memory requirements of the analysis. Additionally, preemption would increase the possible behaviour of the system and, consequently, the number of states of its underlying stochastic process.

Because the number of states grows rapidly even with our state reduction approach and each state has to store its probability density function, the memory space required to store the whole process can become prohibitively large. Our solution to master memory complexity is to perform the stochastic process construction and analysis simultaneously. As each arrow updates the time probability density z of the state it leads to, the process has to be constructed in topological order. The result of this procedure is that the process is never stored entirely in memory but rather that a *sliding window of states* is used for analysis. For the example in Figure 4.4, the construction starts with state s_1. After its next states (s_2 and s_3) are created, their corresponding transition probabilities determined and the possible deadline miss probabilities accounted for, state s_1 can be removed from memory. Next, one of the states s_2 and s_3 is taken as current state, let us consider state s_2. The procedure is repeated, states s_4, s_5 and s_6 are created and state s_2 removed. At this moment, one would think that any of the states s_3, s_4, s_5, and s_6 can be selected for continuation of the analysis. However, this is not the case, as not all the information needed in order to handle states s_5 and s_6 are computed. More exactly, the arcs emerging from states s_3 and s_4 have not yet been created. Thus, only states s_3 and s_4 are possible alternatives for the continuation of the analysis in topological order. The next section discusses the criteria for selection of the correct state to continue with.

4.2.2 Memory Efficient Analysis Method

As shown in the example in Section 4.2.1, only a sliding window of states is simultaneously kept in memory. All states belonging to the

sliding window are stored in a priority queue. Once a state is extracted from this queue and its information processed, it is eliminated from the memory. The key to the process construction in topological order lies in the order in which the states are extracted from this queue. First, observe that it is impossible for an arc to lead from a state with a PMI number u to a state with a PMI number v such that $v < u$ (there are no arcs back in time). Hence, a first criterion for selecting a state from the queue is to select the one with the smallest PMI number. A second criterion determines which state has to be selected out of those with the same PMI number. Note that inside a PMI no new task instantiation can arrive, and that the task ordering according to their priorities is unchanged. Thus, it is impossible that the next state s_k of a current state s_j would be one that contains waiting tasks of higher priority than those waiting in s_j. Hence, the second criterion reads: among states with the same PMI, one should choose the one with the waiting task of highest priority. Figure 4.5 illustrates the algorithm on the example given in Section 4.2.1 (Figure 4.4). The shades of the states denote their PMI number. The lighter the shade, the smaller the PMI number. The numbers near the states denote the sequence in which the states are extracted from the queue and processed.

4.2.3 Multiple Simultaneously Active Instantiations of the Same Task Graph

The examples considered so far dealt with applications where at most one active instance of each task graph is allowed at any moment of time ($b_i = 1, 1 \leq i \leq g$).

In order to illustrate the construction of the stochastic process in the case $b_i > 1$, when several instantiations of a task graph Γ_i may exist at the same time in the system, let us consider an application consisting of two independent tasks, τ_1 and τ_2, with periods 2 and 4 respectively. $LCM = 4$ in this case. The tasks are scheduled according to a rate monotonic (RM) policy [LL73]. At most one active instantiation of τ_1 is tolerated in the system at a certain time ($b_1 = 1$) and at most two concurrently active instantiations of τ_2 are tolerated in the system ($b_2 = 2$).

Figure 4.6 depicts a part of the stochastic process underlying this example. It is constructed using the procedure sketched in Sections 4.2.1 and 4.2.2. The state indexes show the order in which the states were analysed (extracted from the priority queue mentioned in Section 4.2.2).

Let us consider state $s_6 = (\tau_2, \varnothing, [2, 4))$, i.e. the instantiation of τ_2 that arrives at time moment 0 has been started at a moment inside the PMI $[2, 4)$ and there have not been any ready tasks at the start time of τ_2. Let us assume that the finishing time of τ_2 lies past the $LCM = 4$. At time moment 4, a new instantiation of τ_2 arrives and the running instantiation is *not* discarded, as $b_2 = 2$. On one hand, if the finishing time of the running instantiation belongs to the interval $[6, 8)$, the system performs the transition $s_6 \rightarrow s_{14}$ (Figure 4.6). If, on the other hand, the running instantiation attempts to run past the time moment 8, then at this time moment a *third* instantiation of τ_2 would require service from the system and, therefore, the running task (the oldest instantiation of τ_2) is eliminated from the system. The transition $s_6 \rightarrow s_{19}$

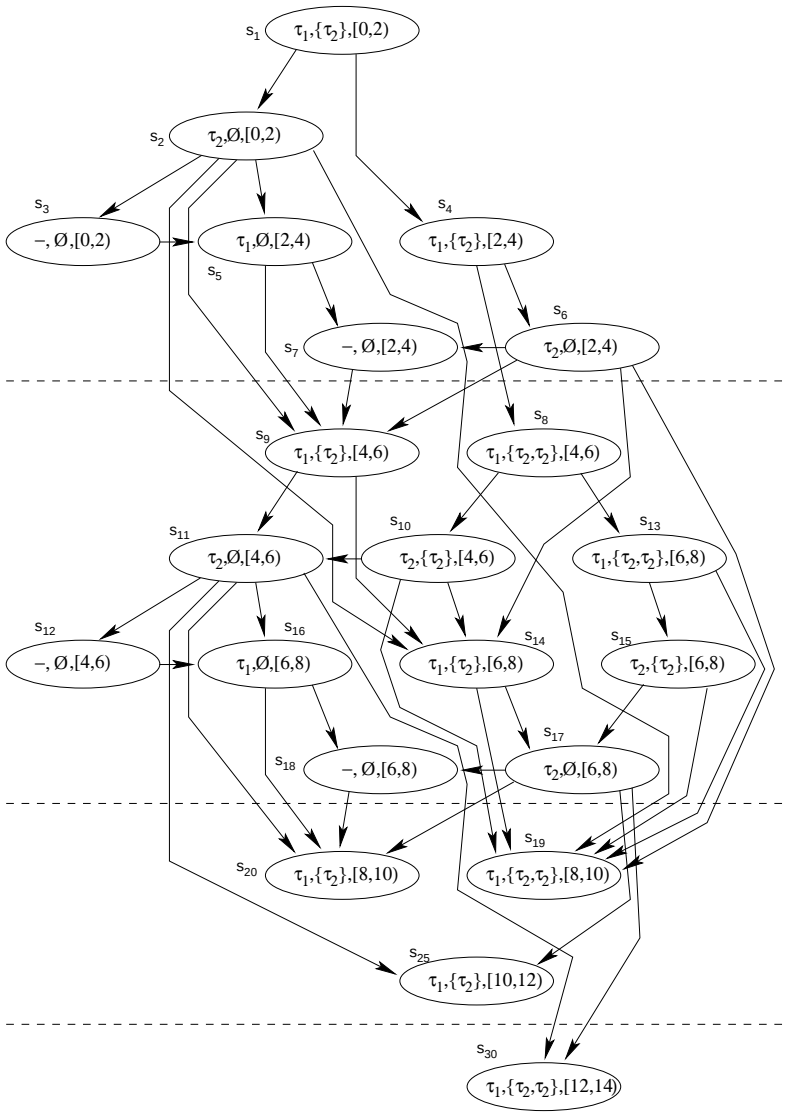


Figure 4.6: Part of the stochastic process underlying the example application

in the stochastic process in Figure 4.6 corresponds to this latter case. We observe that when a task execution spans beyond the time moment LCM, the resulting state is not unique. The system does not behave as if it has just been restarted at time moment LCM, and, therefore, the intervals $[k \cdot LCM, (k+1) \cdot LCM)$, $k \in \mathbb{N}$, are not statistically equivalent to the interval $[0, LCM)$. Hence, it is not sufficient to analyse the system over the interval $[0, LCM)$ but rather over several consecutive intervals of length LCM.

Let an interval of the form $[k \cdot LCM, (k+1) \cdot LCM)$ be called the *hyperperiod* k and denoted H_k. $H_{k'}$ is a *lower* hyperperiod than H_k ($H_{k'} < H_k$) if $k' < k$. Consequently, H_k is a *higher* hyperperiod than $H_{k'}$ ($H_k > H_{k'}$) if $k > k'$.

For brevity, we say that a state s belongs to a hyperperiod k (denoted $s \in H_k$) if its PMI field is a subinterval of the hyperperiod k. In our example, three hyperperiods are considered, $H_0 = [0, 4)$, $H_1 = [4, 8)$, and $H_2 = [8, 12)$. In the stochastic process in Figure 4.6, $s_1, s_2, \ldots, s_7 \in H_0$, $s_8, s_9, \ldots, s_{18} \in H_1$, and $s_{19}, s_{20}, s_{25} \in H_2$ (note that not all states have been depicted in Figure 4.6).

In general, let us consider a state s and let $\mathcal{P}_s$ be the set of its predecessor states. Let k denote the *order* of the state s defined as the lowest hyperperiod of the states in $\mathcal{P}_s$ ($k = \min\{j : s' \in H_j, s' \in \mathcal{P}_s\}$). If $s \in H_k$ and s is of order k' and $k' < k$, then s is a *back state*. In our example, s_8, s_9, s_{14}, and s_{19} are back states of order 0, while s_{20}, s_{25} and s_{30} are back states of order 1.

Obviously, there cannot be any transition from a state belonging to a hyperperiod H to a state belonging to a lower hyperperiod than H ($s \rightarrow s', s \in H_k, s' \in H_{k'} \Rightarrow H_k \leq H_{k'}$). Consequently, the set $\mathcal{S}$ of all states belonging to hyperperiods greater or equal to H_k can be constructed from the back states of an order smaller than k. We say that $\mathcal{S}$ is *generated* by the aforementioned back states. For example, the set of all states $s_8, s_9, \ldots, s_{18} \in H_1$ can be derived from the back states s_8, s_9, and s_{14} of order 0. The intuition behind this is that back states inherit all the needed information across the border between hyperperiods.

Before continuing our discussion, we have to introduce the notion of *similarity* between states. We say that two states s_i and s_j are similar ($s_i \sim s_j$) if all the following conditions are satisfied:

1. The task that is running in s_i and the one in s_j are the same,
2. The multiset of ready tasks in s_i and the one in s_j are the same,
3. The PMIs in the two states differ only by a multiple of LCM, and
4. $z_i = z_j$ (z_i is the probability density function of the times when the system takes a transition to s_i).

Let us consider the construction and analysis of the stochastic process, as described in Sections 4.2.1 and 4.2.2. Let us consider the moment x, when the last state belonging to a certain hyperperiod H_k has been eliminated from the sliding window. R_k is the set of back states stored in the sliding window at the moment x. Let the analysis proceed with the states of the hyperperiod H_{k+1} and let us consider the moment y when the last state belonging to H_{k+1} has been eliminated from the sliding window. Let R_{k+1} be the set of back states stored in the sliding window at moment y.

If the sets R_k and R_{k+1} contain pairwise similar states, then it is guaranteed that R_k and R_{k+1} generate identical stochastic processes

during the rest of the analysis procedure (as stated, at a certain moment the set of back states unambiguously determines the rest of the stochastic process). In our example, $R_0 = \{s_8, s_9, s_{14}, s_{19}\}$ and $R_1 = \{s_{19}, s_{20}, s_{25}, s_{30}\}$. If $s_8 \sim s_{19}$, $s_9 \sim s_{20}$, $s_{14} \sim s_{25}$, and $s_{19} \sim s_{30}$ then the analysis process may stop as it reached convergence.

Consequently, the analysis proceeds by considering states of consecutive hyperperiods until the information captured by the back states in the sliding window does not change any more. Whenever the underlying stochastic process has a steady state, this steady state is guaranteed to be found.

4.2.4 Construction and Analysis Algorithm

The analysis is performed in two phases:

1. Divide the interval $[0, LCM)$ in PMIs,
2. Construct the stochastic process in topological order and analyse it.

Let A denote the set of task arrivals in the interval $[0, LCM]$, i.e. $A = \{x | 0 \leq x \leq LCM, \exists 1 \leq i \leq N, \exists k \in \mathbb{N} : x = k\pi_i\}$. Let D denote the set of deadlines in the interval $[0, LCM]$, i.e. $D = \{x | 0 \leq x \leq LCM, \exists 1 \leq i \leq N, \exists k \in \mathbb{N} : x = k\pi_i + \delta_i\}$. The set of PMIs of $[0, LCM)$ is $\{[a, b) | a, b \in A \cup D \wedge \nexists x \in (A \cup D) \cap (a, b)\}$. If PMIs of a higher hyperperiod H_k, $k > 0$, are needed during the analysis, they are of the form $[a + k \cdot LCM, b + k \cdot LCM)$, where $[a, b)$ is a PMI of $[0, LCM)$.

The algorithm proceeds as discussed in Sections 4.2.1, 4.2.2 and 4.2.3. An essential point is the construction of the process in topological order, which requires only parts of the states to be stored in memory at any moment. The algorithm for the stochastic process construction and analysis is depicted in Figure 4.7.

A global priority queue stores the states in the sliding window. The state priorities are assigned as shown in Section 4.2.2. The initial state of the stochastic process is put in the queue. The explanation of the algorithm is focused on the `construct_and_analyse` procedure (lines 9–27). Each invocation of this procedure constructs and analyses the part of the underlying stochastic process that corresponds to one hyperperiod H_k. It starts with hyperperiod H_0 ($k = 0$). The procedure extracts one state at a time from the queue. Let $s_j = (\tau_i, W_i, pmi_i)$ be such a state. The probability density of the time when a transition occurs *to* s_j is given by the function z_j. The priority scheme of the priority queue ensures that s_j is extracted from the queue only after *all* the possible transitions to s_j have been considered, and thus z_j contains accurate information. In order to obtain the probability density of the time when task τ_i completes its execution, the probability density of its starting time (z_j) and the ETPDF of τ_i (ϵ_i) have to be convoluted. Let ξ be the probability density resulting from the convolution.

Figure 4.8 presents an algorithmic description of the procedure $next\_state$. Based on ξ, the finishing time PDF of task τ_i if task τ_i is never discarded, we compute the maximum execution time of task τ_i, max_exec_time. max_time is the minimum between max_exec_time and the time at which task τ_i would be discarded. PMI will then denote the set of all PMIs included in the interval between the start of the PMI in which task τ_i started to run and max_time. Task τ_i could, in principle, complete its execution during any of these PMIs. We consider each PMI as being the one in which task τ_i finishes its execution. A new

```
(1)  divide [0, LCM) in PMIs;
(2)  put first state in the
     priority queue pqueue;
(3)  R_old = ∅;                          // R_old is the set of densities z
                                            of the back states after itera-
                                            tion k
(4)  (R_new, Missed) =
       = construct_and_analyse();        // Missed is the set of expected
                                            deadline miss ratios
(5)  do
(6)    R_old = R_new;
(7)    (R_new, Missed) =
         = construct_and_analyse();
(8)  while R_new ≠ R_old;

     construct_and_analyse:
(9)  while ∃s ∈ pqueue such that
       s.pmi ≤ pmi_no do
(10)   s_j = extract state from pqueue;
(11)   τ_i = s_j.running;                // first field of the state
(12)   ξ = convolute(ε_i, z_j);
(13)   nextstatelist = next_states(s_j); // consider task dependencies!
(14)   for each s_u ∈ nextstatelist do
(15)     compute the probability
           of the transition from
           s_j to s_u using ξ;
(16)     update deadline miss
           probabilities Missed;
(17)     update z_u;
(18)     if s_u ∉ pqueue then
(19)       put s_u in the pqueue;
(20)     end if;
(21)     if s_u is a back state
           and s_u ∉ R_new then
(22)       R_new = R_new ∪ {s_u};
(23)     end if;
(24)   end for;
(25)   delete state s_j;
(26) end while;
(27) return (R_new, Missed);
```

Figure 4.7: Construction and analysis algorithm

next_states($s_j = (\tau_i, W_i, t_i)$**):**

(1) $nextstates = \varnothing$;

(2) $max\_exec\_time = sup\{t : \xi(t) > 0\}$; // the largest finishing time of τ_i

(3) $max\_time = min\{max\_exec\_time, discarding\_time_i\}$ // the minimum between finishing time and discarding time of τ_i

(4) $PMI = \{[lo_p, hi_p) \in PMIs : lo_p \geq t_i \wedge hi_p \leq max\_time\}$ // the set of PMIs included in the interval $[t_i, max\_time]$

(5) **for each** $[lo_p, hi_p) \in PMI$ **do**

(6) $Arriv = \{\tau \in T : \tau$ arrived in the interval $[t_i, hi_p)\}$;

(7) $Discarded = \{\tau \in W_i : \tau$ was discarded in the interval $[t_i, hi_p)\}$;

(8) $Enabled = \{\tau \in T : \tau$ becomes ready to execute as an effect of τ_i's completion$\}$;

(9) $W = (W_i \backslash Discarded) \cup Enabled \cup \{\tau \in Arriv : {}^\circ\tau = \varnothing\}$; // add the newly arrived tasks with no predecessors, as they are ready to execute, and the newly enabled ones

(10) select the new running task τ_u from W based on the scheduling policy

(11) $W_u = W \backslash \{\tau_u\}$;

(12) add $(\tau_u, W_u, [lo_p, hi_p))$ to $nextstatelist$;

(13) **end for**;

(14) **return** $nextstates$;

Figure 4.8: $next\_states$ procedure

underlying stochastic process state corresponds to each of these possible finishing PMIs. For each PMI, we determine the multiset $Arriv$ of newly arrived tasks while task τ_i is executing. Also, we determine the multiset $Discarded$ of those tasks that are ready to execute when task τ_i starts, but are discarded in the mean time, as the execution of task τ_i spans beyond their deadlines. Once task τ_i completes its execution, some of its successor tasks may become ready to execute. The successor tasks, which become ready to execute as a result of task τ_i's completion, form the set $Enabled$. The new multiset of ready tasks, W, is the union of the old multiset of ready tasks except the ones that are discarded during the execution of task τ_i, $W_i \backslash Discarded$, and the set $Enabled$ and those newly arrived tasks that have no predecessor and therefore are immediately ready to run. Once the new set of ready tasks is determined, the new running task τ_u is selected from multiset W based on the scheduling policy of the application. A new stochastic process state $(\tau_u, W \backslash \{\tau_u\}, [lo_p, hi_p))$ is constructed and added to the list of next states.

The probability densities z_u of the times a transition to $s_u \in nextstatelist$ is taken are updated based on ξ. The state s_u is then added to the priority queue and s_j removed from memory. This procedure is repeated until there is no task instantiation that starts its execution in hyperperiod H_k (until no more states in the queue have their PMI field in the range $k \cdot pmi\_no, \ldots, (k+1) \cdot pmi\_no$, where pmi_no is the number of PMIs between 0 and LCM). Once such a situation is reached, partial results, corresponding to the hyperperiod H_k are available and the `construct_and_analyse` procedure returns. The `construct_and_analyse` procedure is repeated until the set of back states R does not change any more.

4.3 Experimental Results

The most computation intensive part of the analysis is the computation of the convolutions $z_i * \epsilon_j$. In our implementation we used the FFTW library [FJ98] for performing convolutions based on the Fast Fourier Transform. The number of convolutions to be performed equals the number of states of the stochastic process. The memory required for analysis is determined by the maximum number of states in the sliding window. The main factors on which the size of the stochastic process depends are LCM (the least common multiple of the task periods), the number of PMIs, the number of tasks N, the task dependencies, and the maximum allowed number of concurrently active instantiations of the same task graph.

As the selection of the next running task is unique, given the pending tasks and the time moment, the particular scheduling policy has a reduced impact on the process size. Hence, we use the non-preemptive EDF scheduling policy in the experiments below. On the other hand, the task dependencies play a significant role, as they strongly influence the set of ready tasks and, by this, the process size.

The ETPDFs are randomly generated. An interval $[Emin, Emax]$ is divided into smaller intervals. For each of the smaller intervals, the ETPDF has a constant value, different from the value over other intervals. The curve shape has of course an influence on the final result of the analysis, but it has little or no influence on the analysis time and

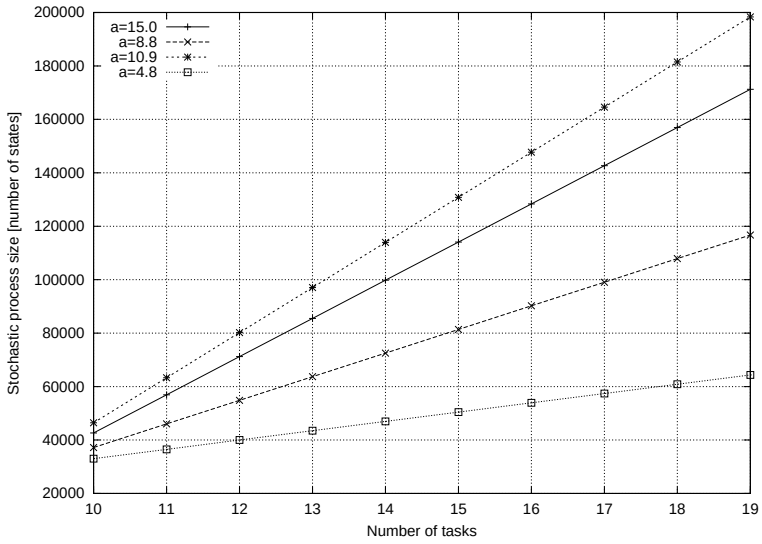


Figure 4.9: Stochastic process size vs. number of tasks

memory consumed by the analysis itself. The interval length $Emax - Emin$ influences the analysis time and memory, but only marginally.

The periods are randomly picked from a pool of periods with the restriction that the period of task τ has to be an integer multiple of the periods of the predecessors of task τ. The pool comprises periods in the range $2, 3, \ldots, 24$. Large prime numbers have a lower probability to be picked, but it occurs nevertheless.

In the following, we report on six sets of experiments. The first four investigate the impact of the enumerated factors (LCM, the number N of tasks, the task dependencies, the maximum allowed number of concurrently active instantiations of the same task graph) on the analysis complexity. The fifth set of experiments considers the rejection late task policy and investigates its impact on the analysis complexity. The sixth experiment is based on a real-life example from the area of telecommunication systems.

The aspects of interest were the stochastic process size, as it determines the analysis execution time, and the maximum size of the sliding window, as it determines the memory space required for the analysis. Both the stochastic process size and the maximum size of the sliding window are expressed in number of states. All experiments were performed on an UltraSPARC 10 at 450 MHz.

4.3.1 Stochastic Process Size as a Function of the Number of Tasks

In the first set of experiments we analysed the impact of the number of tasks on the process size. We considered task sets of 10 to 19 independent tasks. LCM, the least common multiple of the task periods, was 360 for all task sets. We repeated the experiment four times for average values of the task periods $a = 15.0$, 10.9, 8.8, and 4.8 (keeping $LCM = 360$). The results are shown in Figure 4.9. Figure 4.10 depicts the maximum size of the sliding window for the same task sets. As it can be seen from the diagram, the increase, both of the process size and

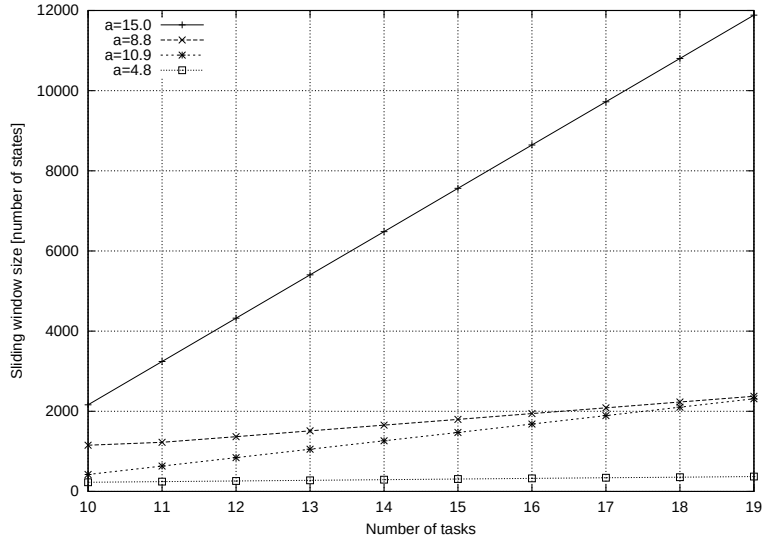


Figure 4.10: Size of the sliding window of states vs. number of tasks

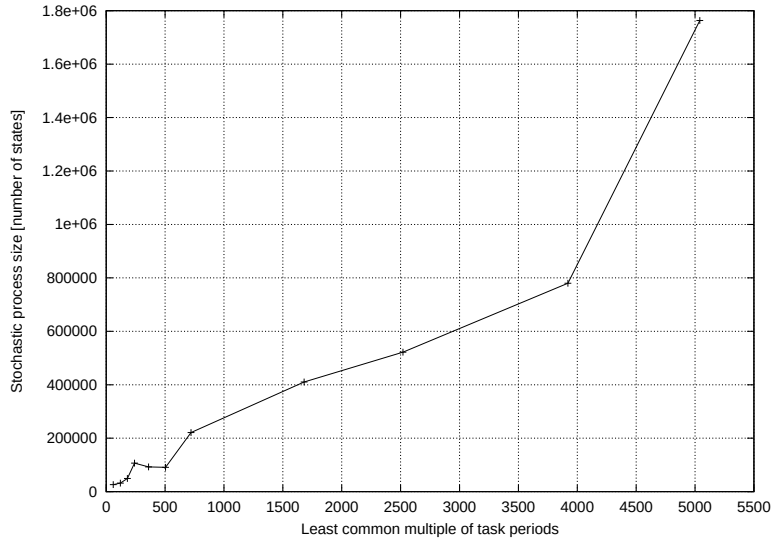


Figure 4.11: Stochastic process size vs. application period LCM

of the sliding window, is linear. The steepness of the curves depends on the task periods (which influence the number of PMIs). It is important to notice the big difference between the process size and the maximum number of states in the sliding window. In the case of 9 tasks, for example, the process size is between 64356 and 198356 while the dimension of the sliding window varies between 373 and 11883 (16 to 172 times smaller). The reduction factor of the sliding window compared to the process size was between 15 and 1914, considering all our experiments.

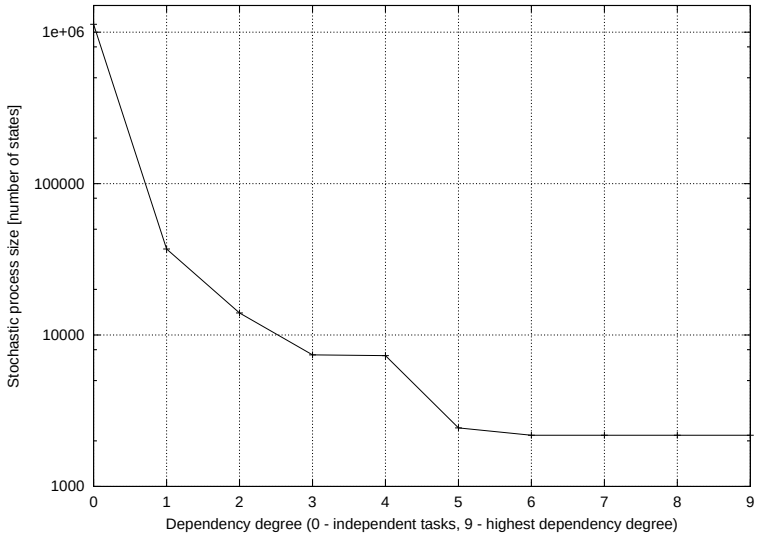


Figure 4.12: Stochastic process size vs. task dependency degree

4.3.2 Stochastic Process Size as a Function of the Application Period

In the second set of experiments we analysed the impact of the application period LCM (the least common multiple of the task periods) on the process size. We considered 784 sets, each of 20 independent tasks. The task periods were chosen such that LCM takes values in the interval $[1, 5040]$. Figure 4.11 shows the variation of the average process size with the application period.

4.3.3 Stochastic Process Size as a Function of the Task Dependency Degree

With the third set of experiments we analysed the impact of task dependencies on the process size. A task set of 200 tasks with strong dependencies (28000 arcs) among the tasks was initially created. The application period LCM was 360. Then 9 new task graphs were successively derived from the first one by uniformly removing dependencies between the tasks until we finally got a set of 200 independent tasks. The results are depicted in Figure 4.12 with a logarithmic scale for the y axis. The x axis represents the degree of dependencies among the tasks (0 for independent tasks, 9 for the initial task set with the highest amount of dependencies).

As mentioned, the execution time for the analysis algorithm strictly depends on the process size. Therefore, we showed all the results in terms of this parameter. For the set of 200 independent tasks used in this experiment (process size 1126517) the analysis time was 745 seconds. In the case of the same 200 tasks with strong dependencies (process size 2178) the analysis took 1.4 seconds.

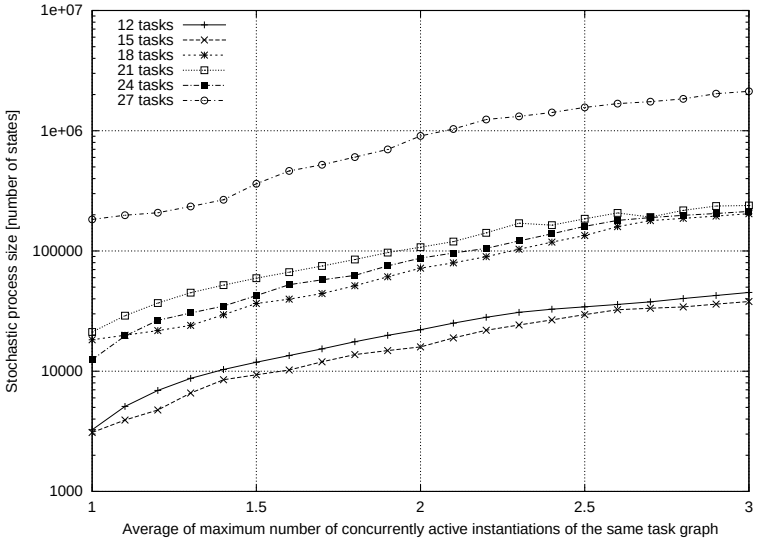


Figure 4.13: Stochastic process size vs. average number of concurrently active instantiations of the same task graph

4.3.4 Stochastic Process Size as a Function of the Average Number of Concurrently Active Instantiations of the Same Task Graph

In the fourth set of experiments, the impact of the average number of concurrently active instantiations of the same task graph on the stochastic process size was analysed. 18 sets of task graphs containing between 12 and 27 tasks grouped in 2 to 9 task graphs were randomly generated. Each task set was analysed between 9 and 16 times considering different upper bounds for the maximum allowed number of concurrently active task graph instantiations. These upper bounds ranged from 1 to 3. The results were averaged for the same number of tasks. The dependency of the underlying stochastic process size as a function of the average of the maximum allowed number of instantiations of the same task graph that are concurrently active is plotted in Figure 4.13. Note that the y-axis is logarithmic. Different curves correspond to different sizes of the considered task sets. It can be observed that the stochastic process size is approximately linear in the average of the maximum allowed number of concurrently active instantiations of the same task graph.

4.3.5 Rejection versus Discarding

As formulated in Section 3.2.6, the discarding policy specifies that the oldest instantiation of task graph Γ_i is eliminated from the system when there are b_i concurrently active instantiations of Γ_i in the system, and a new instantiation of Γ_i demands service. Sometimes, such a strategy is not desired, as the oldest instantiation might have been very close to finishing, and by discarding it, the invested resources (time, memory, bandwidth, etc.) are wasted.

Therefore, our problem formulation has been extended to support a late task policy in which, instead of discarding the oldest instantiation

of Γ_i, the newly arrived instantiation is denied service (rejected) by the system.

In principle, the rejection policy is easily supported by only changing the `next_states` procedure in the algorithm presented in Section 4.2.4. However, this has a strong impact on the analysis complexity as shown in Table 4.1. The significant increase in the stochastic process size (up to two orders of magnitude) can be explained considering the following example. Let s be the stochastic process state under analysis, let τ_j belonging to task graph Γ_i be the task running in s and let us consider that there are b_i concurrently active instantiations of Γ_i in the system. The execution time of τ_j may be very large, spanning over many PMIs. In the case of discarding, it was guaranteed that τ_j will stop running after at most $b_i \cdot \pi_{\Gamma_i}$ time units, because at that time moment it would be eliminated from the system. Therefore, when considering the discarding policy, the number of next states of a state s is upper bounded. When considering the rejection policy, this is not the case any more.

Moreover, let us assume that b_i instantiations of the task graph Γ_i are active in the system at a certain time. In the case of discarding, capturing this information in the system state is sufficient to unambiguously identify those b_i instantiations: they are the last b_i that arrived, because always the oldest one is discarded. For example, the two ready instantiations of τ_2 in the state $s_{13} = (\tau_1, \{\tau_2, \tau_2\}, [6, 8))$ in Figure 4.6 are the ones that arrived at the time moments 0 and 4. However, when the rejection policy is deployed, just specifying that b_i instantiations are in the system is not sufficient for identifying them. We will illustrate this by means of the following example. Let $b_i = 2$, and let the current time be $k\pi_{\Gamma_i}$. In a first scenario, the oldest instantiation of Γ_i, which is still active, arrived at time moment $(k-5)\pi_{\Gamma_i}$ and it still runs. Therefore, the second oldest instantiation of Γ_i is the one that arrived at time moment $(k-4)\pi_{\Gamma_i}$ and all the subsequent instantiations were rejected. In a second scenario, the instantiation that arrived at time moment $(k-5)\pi_{\Gamma_i}$ completes its execution shortly before time moment $(k-1)\pi_{\Gamma_i}$. In this case, the instantiations arriving at $(k-3)\pi_{\Gamma_i}$ and $(k-2)\pi_{\Gamma_i}$ were rejected but the one arriving at $(k-1)\pi_{\Gamma_i}$ was not. In both scenarios, the instantiation arriving at $k\pi_{\Gamma_i}$ is rejected, as there are two concurrently active instantiations of Γ_i in the system, but these two instantiations cannot be determined without extending the definition of the stochastic process state space. Extending this space with the task graph arrival times is partly responsible for the increase in number of states of the underlying stochastic process.

The fifth set of experiments reports on the analysis complexity when the rejection policy is deployed. 101 task sets of 12 to 27 tasks grouped in 2 to 9 task graphs were randomly generated. For each task set two analyses were performed, one considering the discarding policy and the other considering the rejection policy. The results were averaged for task sets with the same cardinality and shown in Table 4.1.

4.3.6 Encoding of a GSM Dedicated Signalling Channel

Finally, we present an example from industry, in particular the mobile communication area.

Tasks	Average stochastic process size [number of states]		Relative increase
	Discarding	Rejection	
12	2223.52	95780.23	42.07
15	7541.00	924548.19	121.60
18	4864.60	364146.60	73.85
21	18425.43	1855073.00	99.68
24	14876.16	1207253.83	80.15
27	55609.54	5340827.45	95.04

Table 4.1: Discarding compared to rejection

Mobile terminals access telecommunication networks via a radio interface. An interface is composed of several *channels*. As the electromagnetic signals on the radio interface are vulnerable to distortion due to interference, fading, reflexion, etc., sophisticated schemes for error detection and correction are deployed in order to increase the reliability of the channels of the radio interface.

On the network side, the device responsible with radio transmission and reception and also with all signal processing specific to the radio interface is the *base transceiver station (BTS)*. The chosen demonstrator application is the baseband processing of the stand-alone dedicated control channel (SDCCH) of the Global System for Mobile Communication [ETS]. It represents a rather complex case, making use of all of the stages of baseband processing.

The task graphs that model the downlink part (BTS to mobile station) of the GSM SDCCH are shown in Figure 4.14. Every four frame periods, i.e. every $240/13$ms ≈ 18.46ms, a block of 184 bits, specified in GSM 05.03, requires transmission. This block is denoted $block1$ in Figure 4.14. The block is processed by a so-called FIRE encoder that adds 40 parity bits to $block1$. The FIRE encoder and its polynomial are specified in GSM 05.03, section 4.1.2. The result of the FIRE encoding is $block2$, a 224 bits block. Four zero bits are appended to $block2$ by the tailer as specified in the aforementioned GSM document. The result is $block3$, a 228 bits block. The 228 bits of $block3$ are processed by a convolutional encoder specified in GSM 05.03, section 4.1.3, where the generating polynomials are given. The result of the convolutional encoding is $block4$, a 456 bit block. $Block5_1$, 5_2, 5_3, and 5_4, each 114 bits long, result from the interleaving of $block4$ as specified by GSM 05.03, section 4.1.4. Depending on how the establishment of the channel was negotiated between the mobile station and the network, the communication may or may not be encrypted. In case of encryption, blocks 6_1 to 6_4 result from blocks 5_1 to 5_4 respectively, as specified in GSM 03.20, annex 3, sections A3.1.2 and A3.1.3. $Block6_1$, 6_2, 6_3, and 6_4 are then assembled as specified by GSM 05.03, section 4.1.5 and GSM 05.02, section 5.2.3. The assembling is done using a training sequence TS. TS is a 26 bit array and is one of the 8 training sequences of GSM, specified in GSM 05.02, section 5.2.3. The assembling of blocks 6_1 to 6_4 results in bursts 6_1 to 6_4, each of these bursts being 148 bits long. Bursts 7_1 to 7_4 result from the modulation of bursts 6_1 to 6_4 respectively. Bursts 8_1 to 8_4 are radio bursts modulated on frequencies $freq1$ to $freq4$ respectively. $Freq1$, $freq2$, $freq3$, and $freq4$ are integers, maximum 6 bit long in GSM900, that indicate the frequency to be used for sending a burst.

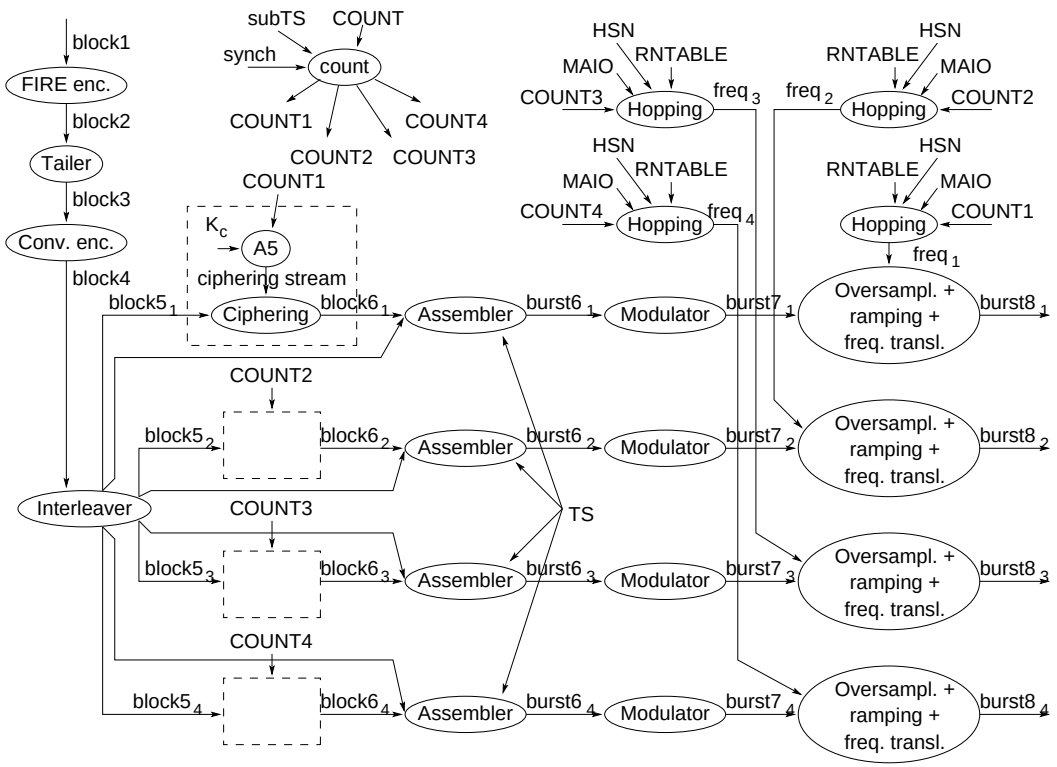


Figure 4.14: Encoding of a GSM dedicated signalling channel

They are computed as specified by GSM 05.02, section 6.2.3. Their computation makes use of the 6 bit integers MAIO (mobile allocation index offset) and HSN (hopping sequence number), and of RNTABLE, a vector of 114 integers of 7 bits each, specified by GSM 05.02, section 6.2.3. $COUNT$ is the current frame number while $COUNT1$ to 4 are numbers of the frames in which the four bursts will be sent on the radio interface. $COUNT1$ to 4 are obtained by task "count" from the subtimeslot number of the SDC channel ($subTS$), from a clock tick $synch$, and from the current frame number $COUNT$.

The graph in Figure 4.14 contains many functional units of fine granularity, which could induce high communication overhead. In order to reduce the overhead, some replicated units could be collapsed, other could be merged together. The modified graph is depicted in Figure 4.15. In this case, the A5, ciphering, assembling, modulating, hopping, and oversampling units iterate four times in each activation of the graph. Merging the interleaver and the assembler leads to the modification of the algorithms of the interleaver and the ciphering unit. The ciphering unit does not receive 114 bits from the interleaver, but 148 bits, structured in the form $3 + 57 + 1 + 26 + 1 + 57 + 3$. The ciphering unit performs then a XOR between the 114 bits ciphering stream and the two 57-bit fields of the received block, leaving the remaining $3 + 1 + 26 + 1 + 3$ bits untouched.

The whole application runs on a single DSP processor and the tasks are scheduled according to fixed priority scheduling. The FIRE encoding task has a period of $240/13 \approx 18.46$ms.[2] FIRE encoding, convolutional encoding and interleaving are activated once every task graph instantiation, while the ciphering, A5, modulator, hopping, and oversampling tasks are activated four times every task graph instantiation. The end-to-end deadline of the task graph is equal to its period, i.e. $240/13$ms.

In this example, there are two sources of variation in execution times. The modulating task has both data and control intensive behaviour, which can cause pipeline hazards on the deeply pipelined DSP it runs on. Its execution time probability density is derived from the input data streams and measurements. Another task will implement a ciphering unit. Due to the lack of knowledge about the deciphering algorithm A5 (its specification is not publicly available), the ciphering task execution time is considered to be uniformly distributed between an upper and a lower bound.

When two channels are scheduled on the DSP, the ratio of missed deadlines is 0 (all deadlines are met). Considering three channels assigned to the same processor, the analysis produced a ratio of missed deadlines, which was below the one enforced by the required QoS. It is important to note that using a hard real-time model with WCET, the system with three channels would result as unschedulable on the selected DSP. The underlying stochastic process for the three channels had 130 nodes and its analysis took 0.01 seconds. The small number of nodes is caused by the strong harmony among the task periods, imposed by the GSM standard.

[2] We use a time quanta of $1/13$ms in this application such that all task periods may be specified as integers.

Figure 4.15: Encoding of a GSM dedicated signalling channel, reduced architecture

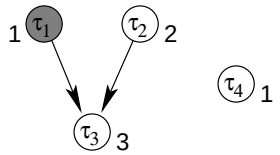
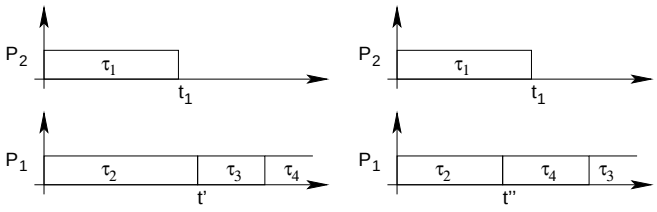


Figure 4.16: Example of multiprocessor application



(a) Scenario 1 (b) Scenario 2

Figure 4.17: Two execution scenarios

4.4 Limitations and Extensions

Although our proposed method is, as shown, efficiently applicable to the analysis of applications implemented on monoprocessor systems, it can handle only small scale multiprocessor applications. This section identifies the causes of this limitation and sketches an alternative approach to handle multiprocessor applications.

When analysing multiprocessor applications, one approach could be to decompose the analysis problem into several subproblems, each of them analysing the tasks mapped on one of the processors. We could attempt to apply the present approach in order to solve each of the subproblems. Unfortunately, in the case of multiprocessors and with the assumption of data dependencies among tasks, this approach cannot be applied. The reason is that the set of ready tasks cannot be determined based solely on the information regarding the tasks mapped on the processor under consideration. To illustrate this, let us consider the example in Figure 4.16. Tasks τ_2, τ_3, and τ_4 are mapped on processor P_1 and task τ_1 is mapped on processor P_2. The numbers near the tasks indicate the task priorities. For simplicity, let us assume that all tasks have the same period π, and hence there is only one priority monotonicity interval $[0, \pi)$. Let us examine two possible scenarios. The corresponding Gantt diagrams are depicted in Figure 4.17. At time moment 0 task τ_1 starts running on processor P_2 and task τ_2 starts running on processor P_1. Task τ_1 completes its execution at time moment $t_1 \in [0, \pi)$. In the first scenario, task τ_2 completes its execution at time moment $t' > t_1$ and task τ_3 starts executing on the processor P_1 at time moment t' because it has the highest priority among the two ready tasks τ_3 and τ_4 at that time. In the second scenario, task τ_2 completes its execution at time moment $t'' < t_1$. Therefore, at time moment t'', only task τ_4 is ready to run and it will start its execution on the processor P_1 at that time. Thus, the choice of the next task to run is not independent of the time when the running task completes its execution inside a PMI. This

makes the concept of PMIs unusable when looking at the processors in isolation.

An alternative approach would be to consider all the tasks and to construct the global state space of the underlying stochastic process accordingly. In principle, the approach presented in the previous sections could be applied in this case. However, the number of possible execution traces, and implicitly the stochastic process, explodes due to the parallelism provided by the application platform. As shown, the analysis has to store the probability distributions z_i for each process state in the sliding window of states, leading to large amounts of needed memory and limiting the appropriateness of this approach to very small multiprocessor applications. Moreover, the number of convolutions $z_i * \epsilon_j$, being equal to the number of states, would also explode, leading to prohibitive analysis times. The next chapter presents an approach that overcomes these problems. However, as opposed to the method presented in this chapter, which produces exact values for the expected deadline miss ratios, the alternative approach generates approximations of the real ratios.

Chapter 5

Analysis of Multiprocessor Systems

The challenge taken in this chapter is to analyse an application running on a multiprocessor system with acceptable accuracy, without the need to explicitly store and compute the memory consuming distributions of the residual execution times of each task in the states of the underlying stochastic process. Also, we would like to avoid the calculation of the computation-intensive convolutions.

We address this problem by using an approximation approach for the task execution time probability distribution functions. Approximating the generalised ETPDFs with weighted sums of convoluted exponential functions leads to approximating the underlying generalised semi-Markov process with a continuous time Markov chain. By doing so, we avoid both the computation of convolutions and the storage of the z_i functions. However, as opposed to the method presented in the previous chapter, which produces exact values for the expected deadline miss ratios, the alternative approach generates approximations of the real ratios.

The approximation of the generalised task execution time probability distributions by weighted sums of convoluted exponential distributions leads to a large continuous time Markov chain. Such a Markov chain is much larger than the stochastic process underlying the system with the real, non-approximated execution times, but, as the state holding times probability distributions are exponential, there is no need to explicitly store their distributions, leading to a much more efficient use of the analysis memory. Moreover, by construction, the Markov chain exhibits regularities in its structure. These regularities are exploited during the analysis such that the infinitesimal generator of the chain is constructed on-the-fly, saving additional amounts of memory. In addition, the solution of the continuous time Markov chain does not imply any computation of convolutions. As a result, multiprocessor applications of realistic size may be analysed with sufficient accuracy. Moreover, by controlling the precision of the approximation of the ETPDFs, the designer may trade analysis resources for accuracy.

5.1 Problem Formulation

The multiprocessor system analysis problem that we solve in this chapter is formulated as follows.

5.1.1 Input

The input of the problem consists of:

- The set of task graphs Γ,
- The set of processors P,
- The mapping Map,
- The set of task periods Π_T and the set of task graph periods Π_Γ,
- The set of task deadlines Δ_T and the set of task graph deadlines Δ_Γ,
- The set of execution time probability density functions ET,
- The late task policy is the discarding policy,
- The set $Bounds = \{b_i \in \mathbb{N}\backslash\{0\} : 1 \leq i \leq g\}$, where b_i is the maximum numbers of simultaneously active instantiations of task graph Γ_i, and
- The scheduling policies on the processing elements and buses.

5.1.2 Output

The results of the analysis are the sets $Missed_T$ and $Missed_\Gamma$ of expected deadline miss ratios for each task and task graph respectively.

5.1.3 Limitations

For now, we restrict our assumptions on the system to the following:

- All tasks belonging to the same task graph have the same period ($\pi_a = \pi_b, \forall \tau_a, \tau_b \in V_i \subset T$, where $\Gamma_i = (V_i, E_i)$ is a task graph),
- The task deadlines (task graph deadlines) are equal to the corresponding task periods (task graph periods) ($\pi_i = \delta_i$, $\forall 1 \leq i \leq N$, and $\pi_{\Gamma_i} = \delta_{\Gamma_i}$, $\forall 1 \leq i \leq g$), and
- The late task policy is the discarding policy.

These restrictions are relaxed in Section 5.9 where we discuss their impact on the analysis complexity.

5.2 Approach Outline

In order to extract the desired performance metrics, the underlying stochastic process corresponding to the application has to be constructed and analysed. Events, such as the arrival of a deadline, represent state transitions in the stochastic process. In order to obtain the long-time average rate of such events, the stationary state probabilities have to be calculated for the stochastic process underlying the system.

The underlying stochastic process is regenerative, i.e. the system behaves probabilistically equivalent in the time intervals between consecutive visits to a regenerative state [Ros70]. Thus, it would

be sufficient to analyse the system in the interval between two consecutive regenerations. However, the subordinated stochastic process (the process between two regeneration points) is a continuous-state time-homogeneous generalised semi-Markov process (GSMP) [She93, Gly89]. Hence, its stationary analysis implies the numerical solution of a system of partial differential equations with complicated boundary conditions [GL94]. This makes the applicability of the GSMP-based analysis limited to extremely small systems.

Because the limitations of the GSMP-based approach, we proceed along a different path, namely the exact analysis of an approximating system. We approximate the generalised probability distributions of task execution times with Coxian probability distributions [Cox55]. The stochastic process that underlies a system with only Coxian probability distributions is a continuous time Markov chain (CTMC) [Lin98], whose steady state analysis implies the solution of a system of linear equations. Albeit theoretically simple, the applicability of this approach, if used directly, is limited by the enormous increase of the number of states of the CTMC relative to the number of states of the stochastic process underlying the application. In order to cope with this increase, we exploit the specific structure of the infinitesimal generator of the CTMC such that we reduce the needed analysis memory by more than one order of magnitude.

The outline of our approach is depicted in Figure 5.1. At step 1, we generate a model of the application as a Concurrent Generalised Petri Net (CGPN) [PST98] (Section 5.3).

At step 2, we construct the tangible reachability graph (TRG) of the CGPN. The TRG is also the marking process, i.e. the stochastic process in which the states represent the tangible markings of the CGPN. The marking process of a CGPN is a generalised semi-Markov process (GSMP) [Lin98] (Section 5.4).

The third step implies the approximation of the arbitrary real-world ETPDFs with Coxian distributions, i.e. weighted sums of convoluted exponential distributions. Some details regarding Coxian distributions and the approximation process follow in Section 5.5.

Directly analysing the GSMP obtained at step 2 is practically impossible (because of time and memory complexity) for even small toy examples, if they are implemented on multiprocessor systems. Therefore, at step 4, the states of this process are substituted by sets of states based on the approximations obtained in the third step. The transitions of the GSMP are substituted by transitions with exponentially distributed firing interval probabilities from the Coxian distributions. What results is a continuous time Markov chain (CTMC), much larger than the GSMP, however easier to analyse. The explanation of this rather counter-intuitive fact is twofold:

- By exploiting regularities in the structure of the CTMC, the elements of its generator matrix can be constructed on-the-fly during the analysis, leading to memory savings.
- Computationally expensive convolutions of probability density functions are avoided by using exponentially distributed ETPDFs.

The construction procedure of the CTMC is detailed in Section 5.6.

As the last step, the obtained CTMC is solved and the performance metrics extracted (Section 5.7).

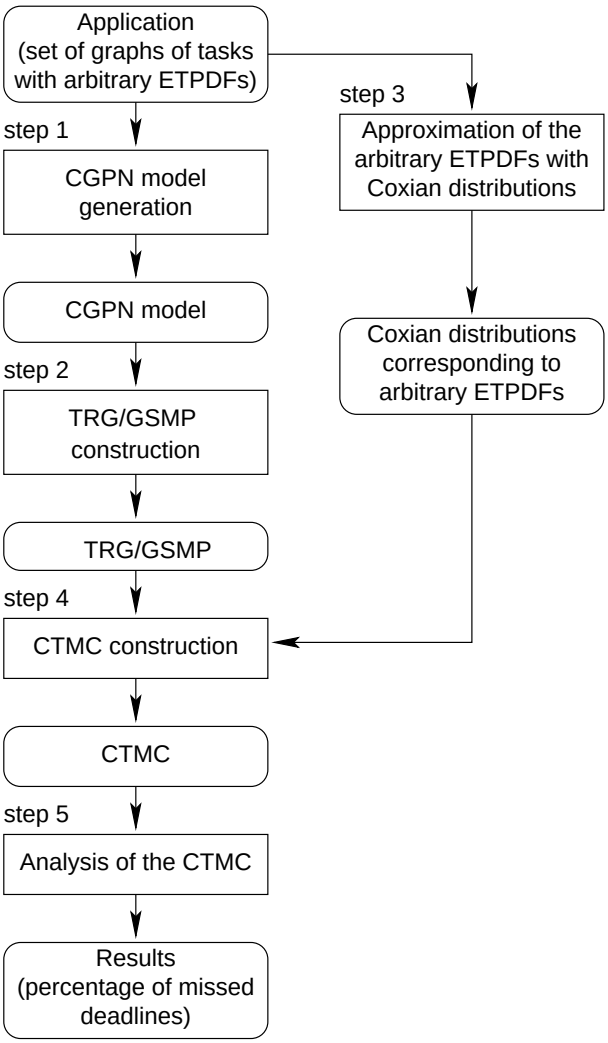


Figure 5.1: Approach outline

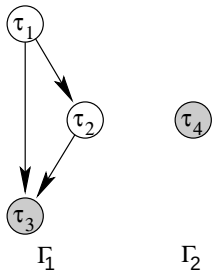


Figure 5.2: Task graphs

5.3 Intermediate Model Generation

As the first step, starting from the task graph model given by the designer, an intermediate model based on Concurrent Generalised Petri Nets (CGPN) [PST98] is generated. Such a model allows an efficient and elegant capturing of the characteristics of the application and of the scheduling policy. It constitutes also an appropriate starting point for the generation of the CTMC, to be discussed in the following sections.

The rest of this section details on the modelling of applications with CGPN. Note that the generation of a CGPN model from the input data of our problem is automatic and its complexity is linear in the number of tasks, and hence negligible relative to the solving of the marking process underlying the system.

5.3.1 Modelling of Task Activation and Execution

We illustrate the construction of the CGPN based on an example. Let us consider the task graphs in Figure 5.2. Tasks τ_1, τ_2 and τ_3 form graph Γ_1 while Γ_2 consists of task τ_4. τ_1 and τ_2 are mapped on processor P_1 and τ_3 and τ_4 on processor P_2. The task priorities are 1, 2, 2, and 1 respectively. The task graph Γ_1 has period π_{Γ_1} and Γ_2 has period π_{Γ_2}. For simplicity, in this example, we ignore the communication tasks.

The CGPN corresponding to the example is depicted in Figure 5.3. Concurrent Generalised Petri Nets are extensions of Generalised Stochastic Petri Nets (GSPN) introduced by Balbo et al. [BCFR87]. CGPNs have two types of transitions, timed (denoted as solid rectangles in Figure 5.3) and immediate (denoted as thin lines). Timed transitions have an associated firing delay, which can be deterministic or stochastic with a given generalised probability distribution function. The firing policy that we consider is *race with enabling policy*, i.e. the elapsed time is kept if the transition stays enabled. Immediate transitions have zero firing delay, i.e. they fire as soon as they are enabled. Immediate transitions have associated priorities, shown as the positive integers that annotate the transitions in the figure. The priorities are used to determine the immediate transition that fires among a set of competitively enabled immediate transitions. Arcs have associated multiplicities, shown as the positive integers that annotate the arcs. A necessary condition for a transition to fire is that the number of tokens in each input place of the transition is equal to or greater than the multiplicity of the arc that connects the corresponding input place to the transition.

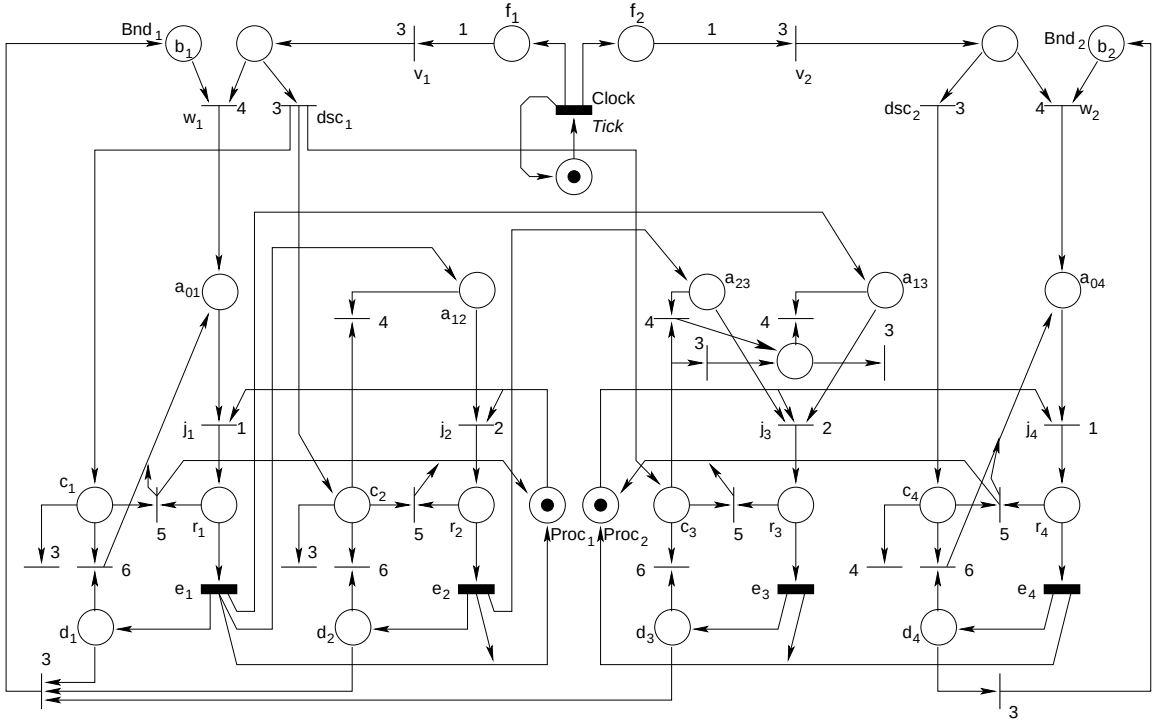


Figure 5.3: CGPN example

The execution of task τ_i, $1 \leq i \leq 4$, is modelled by the place r_i and timed transition e_i. If a timed transition e_i is enabled, it means that an instantiation of the task τ_i is running. The probability distribution of the firing delay of transition e_i is equal to the ETPDF of task τ_i. As soon as e_i fires, it means that the instantiation of τ_i completed execution and leaves the system. The task priorities are modelled by prioritising the immediate transitions j_i.

In our example, the mutual exclusion of the execution of tasks mapped on the same processor is modelled by means of the places $Proc_1$ and $Proc_2$, which correspond to processors P_1 and P_2 respectively. The data dependencies among the tasks are modelled by the arcs $e_2 \rightarrow a_{23}$, $e_1 \rightarrow a_{13}$ and $e_1 \rightarrow a_{12}$.

5.3.2 Modelling of Periodic Task Arrivals

The periodic arrival of graph instantiations is modelled by means of the transition $Clock$ with the deterministic delay $Tick$, as illustrated in Figure 5.3. $Clock$ fires every $Tick$ time units, where $Tick$ is the greatest common divisor of the graph periods. As soon as $Clock$ has fired $\pi_{\Gamma_i}/Tick$ times, the transition v_i fires and a new instantiation of task graph Γ_i demands execution. (In our example, for the simplicity of the illustration, we considered $\pi_{\Gamma_1}/Tick = 1$ and $\pi_{\Gamma_2}/Tick = 1$. This is modelled by specifying an arc multiplicity of 1 and 1 for the arcs $f_1 \rightarrow v_1$ and $f_2 \rightarrow v_2$ respectively.)

5.3.3 Modelling Deadline Misses

In the general case of arbitrary deadlines, the event of an instantiation of a task τ_i (task graph Γ_j) missing its deadline is modelled by the firing of a certain transition (modelling the deadline arrival) in a marking with a certain property (modelling the fact that the considered task (task graph) instantiation has not yet completed).

In the particular case of the task deadline being equal to the task period and to the corresponding task graph period ($\delta_{\tau_i} = \pi_{\tau_i} = \pi_{\Gamma_j}$ if $\tau_i \in V_j$), the arrival of the deadline of any task $\tau_i \in V_j$ is modelled by the firing of v_j, i.e. the arrival of a new instantiation of a task graph Γ_j. The fact that an instantiation of task τ_i has not yet completed its execution is modelled by a marking in which at least one of the places a_{ki} (ready-to-run) or r_i (running) is marked. Hence, the event of an instantiation of τ_i missing its deadline is modelled by the firing of v_j in any of the markings with the above mentioned property.

As explained in Section 3.2.5, the event of an instantiation of task graph Γ_j missing its deadline is equal to the earliest event of a task instantiation of task $\tau_i \in V_j$ missing its deadline δ_{τ_i}. In order to avoid the implied bookkeeping of events, in the case when $\delta_{\tau_i} = \pi_{\tau_i} = \pi_{\Gamma_j}$, $\tau_i \in V_j$, task graph deadline misses have been modelled in the following equivalent but simpler form.

The place Bnd_j is initially marked with b_j tokens, meaning that at most b_j concurrent instantiations of Γ_j are allowed in the system. Whenever a new instantiation of task graph Γ_j is accepted in the system (transition w_j fires), a token is removed from place Bnd_j. Once a task graph instantiation leaves the system (all places d_i are marked, where $\tau_i \in V_j$), a token is added to Bnd_j. Having less than b_j tokens in place Bnd_j indicates that at least one instantiation of task graph Γ_j is

active in the system. Hence, an instantiation of task graph Γ_j misses its deadline if and only if transition v_j fires (denoting the arrival of the deadlines of any task $\tau_i \in V_j$, in the above mentioned particular case) in a CGPN marking with the property that place Bnd_j contains less than b_j tokens.

The modelling of task and task graph deadline misses in the more general cases of $\delta_{\tau_i} = \pi_{\tau_i} \neq \pi_{\Gamma_j}$, and $\delta_{\tau_i} \leq \pi_{\tau_i}$ if $\tau_i \in V_j$ is discussed in Section 5.9.1 and Section 5.9.3.

5.3.4 Modelling of Task Graph Discarding

If Bnd_j contains no tokens at all when v_j fires, then the maximum number of instantiations of Γ_j are already present in the system and, therefore, the oldest one will be discarded. This is modelled by firing the immediate transition dsc_j and marking the places c_i, where one such place c_i corresponds to each task in Γ_j. The token in c_i will attempt to remove from the system an already completed task (a token from d_i), or a running task (a token from r_i), or a ready task (a token from a_{ki}), in this order. The transitions w_j have a higher priority than the transitions dsc_j, in order to ensure that an instantiation of Γ_j is discarded only when Bnd_j contains no tokens (there already are b_j concurrently active instantiations of Γ_j in the system). The structure of the CGPN is such that a newly arrived instantiation is always accepted in the system.

5.3.5 Scheduling Policies

The scheduling policy determines which of the enabled transitions j_i fires. In the case of static priorities, this is easily modelled by assigning the task priorities to the corresponding immediate transitions j_i as is the case of the example in Figure 5.3. In the case of dynamic task priorities, the choice is made based on the global time, which can be deduced from the global marking of the Petri Net. In general, the time left until the deadline of Γ_j is computed by subtracting from the multiplicity of the outgoing arc of f_j (how many $Tick$ units separate consecutive arrivals of Γ_i) the number of tokens of f_j (how many $Tick$ units passed since the last arrival of Γ_j).

In the case of dynamic priorities of tasks, the marking process construction algorithm has to be instructed to choose which transition to fire based on the net marking and not on the transition priorities depicted in the model.

5.4 Generation of the Marking Process

This section discusses step 2 of our approach (Figure 5.1), the generation of the marking process of the Petri Net that models an application.

A tangible marking of a CGPN is a marking in which no immediate transitions are enabled. Such a marking can be directly reached from another tangible marking by firing exactly one timed transition followed by a possibly empty sequence of immediate transition firings, until no more immediate transitions are enabled. The tangible reachability graph (TRG) contains the tangible markings of the Petri net.

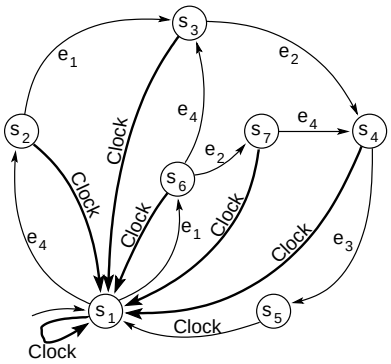


Figure 5.4: Marking process of CGPN in Figure 5.3

Each marking in the TRG corresponds to a state in the underlying stochastic process, also known as the marking process.

Balbo et al. [BCFR87] gave an algorithm for the generation of the tangible reachability graph (TRG) for Generalised Stochastic Petri Nets (GSPN). Even if the Petri Net formalism that we use is an extension to GSPN, the algorithm is applicable nevertheless. In the worst case, the number of nodes in the TRG is exponential in the number of places and number of transitions of the Petri Net.

The tangible reachability graph of the Petri Net in Figure 5.3 is shown in Figure 5.4. The graph in Figure 5.4 is also a graphical representation of the stochastic process of the marking of the net. An edge in the TRG is labelled with the timed transition that triggers the marking change. The thicker arcs correspond to transition firings that model task deadline misses. If we compute the steady-state rates of the firings along the thick arcs, we will be able to obtain the expected deadline miss rates for each task and task graph.

If all timed transitions had exponentially distributed firing delay probabilities, the marking process would be a continuous time Markov chain (CTMC). The computation of its stationary state probability vector would imply the solution of a system of linear equations. As we assume that tasks may have execution times with generalised probability distributions, the marking process is not a CTMC.

Observing that our systems are regenerative[1], a possible solution would be the one proposed by Choi et al. [CKT94]. They introduced Markov Regenerative Stochastic Petri Nets (MRSPN), which allow timed transitions to have firing delays with generalised probability distributions. However, MRSPN have the limitation that at most one transition whose firing delay probability has a generalised distribution is enabled in every marking. In this case, the underlying stochastic process is a Markov regenerative process (MRGP). Choi et al. [CKT94] present a method for the transient analysis of the MRGP corresponding to their marking graph.

One important observation we can make from Figure 5.4 is that in all states, except state s_5, there are more than one simultaneously enabled transitions whose firing delay probabilities have generalised dis-

[1]Let LCM denote the least common multiple of all task periods. The process regenerates itself when both of the following conditions are true: The time becomes $k \cdot LCM$, $k \in \mathbb{N}$ (every $LCM/Tick$ firings of transition $Clock$), and all processors were idle just before this time. In Figure 5.4, such a situation happens every time state s_1 is entered.

tributions. This situation occurs because there are several tasks with non-exponential execution time probability distribution functions that execute concurrently on different processors. Therefore, the process is not a Markov regenerative process with at most one non-exponentially distributed event in each state. Hence, the analysis method of Choi et al. [CKT94] does not apply in our case.

The Concurrent Generalised Petri Net model (CGPN), introduced by Puliafito et al. [PST98], softens the restriction of MRSPNs. Puliafito et al. address the analysis of the marking processes of CGPN with an arbitrary finite number of simultaneously enabled events with generalised probability distribution. Nevertheless, they restrict the system such that all simultaneously enabled transitions get enabled at the same instant. Under this assumption, the marking process is still a MRGP. However, we cannot assume that all tasks that are concurrently executed on a multiprocessor platform start their execution at the same time. Therefore, we remove this restriction placed on CGPN and we will use the term CGPN in a wider sense in the sequel. The marking process of such a CGPN in the wide sense is not necessarily a MRGP.

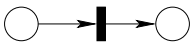
In order to keep the Markovian property, we could expand the state of the marking process to contain the residual firing times of enabled transitions. In this case, the subordinated process, i.e. the process between two regeneration times, is a time-homogeneous generalised semi-Markov process (GSMP) [She93, Gly89]. Hence, its stationary analysis implies the numerical solution of a system of partial differential equations with complicated boundary conditions [GL94]. This makes the applicability of the GSMP-based analysis limited to extremely small systems.

All this leads us to a different approach for solving the marking process. We approximate the generalised probability distribution functions with Coxian distributions [Cox55], i.e. weighted sums of convoluted exponential distribution functions. The resulting process contains transitions with exponentially distributed firing delay probabilities. Hence, it is a continuous time Markov chain (CTMC) that approximates the non-Markovian marking process. The steady-state analysis of the CTMC implies the solving of a system of linear equations. If applied directly, the approach is severely limited by the immense number of states of the approximating CTMC. A key to the efficient analysis of such huge Markov chains is the fact that we observe and exploit a particular structure of the chain, such that its infinitesimal generator may be generated on-the-fly during the analysis and does not need to be stored in memory.

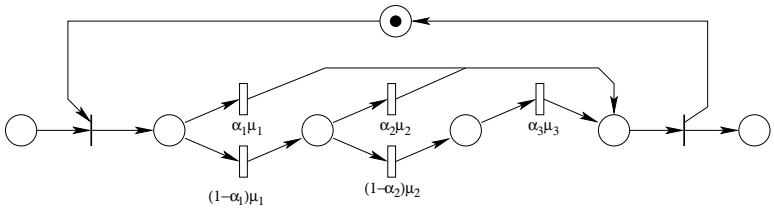
Section 5.5 discusses the approximation of generalised probability distribution functions with Coxian distribution functions. Section 5.6 details on the aforementioned structural properties of the CTMC that allow for its efficient solving.

5.5 Coxian Approximation

Coxian distributions were introduced by Cox [Cox55] in the context of queueing theory. A Coxian distribution of r stages is a weighted sum of convoluted exponential distributions. The Laplace transform of the probability density of a Coxian distribution with r stages is given



(a) A transition with arbitrarily distributed firing delay probability



(b) Subnet modelling a transition whose firing delay probability has a Coxian distribution

Figure 5.5: Coxian approximation with three stages

below:

$$X(s) = \sum_{i=1}^{r} \alpha_i \cdot \prod_{k=1}^{i-1} (1 - \alpha_k) \cdot \prod_{k=1}^{i} \frac{\mu_k}{s + \mu_k}$$

$X(s)$ is a strictly proper rational transform, implying that the Coxian distribution may approximate a fairly large class of arbitrary distributions with an arbitrary accuracy, provided a sufficiently large r.

Figure 5.5 illustrates the way we use Coxian distributions in our approach. Let us consider the timed transition with a certain probability distribution of its firing delay in Figure 5.5(a). This transition can be replaced by the Petri Net in Figure 5.5(b), where hollow rectangles represent timed transitions with exponential firing delay probability distribution. The annotations near those transitions indicate their average firing rate. In this example, three stages have been used for approximation.

Practically, the approximation problem can be formulated as follows: given an arbitrary probability distribution and a certain number of stages r, find $\mu_i, 1 \leq i \leq r$, and $\alpha_i, 1 \leq i \leq r-1$ ($\alpha_r = 1$), such that the quality of approximation of the given distribution by the Coxian distribution with r stages is maximised.

Malhotra and Reibman [MR93] describe a method for parameter fitting that combines moment-matching with least squares fitting for phase approximations, of which Coxians are a subclass. In our approach, because the analytic expression of a Coxian distribution is quite complicated in the time domain, we perform the least squares fitting in the frequency domain. Hence, we minimise the distance between the Fourier transform $X(j\omega)$ of the Coxian distribution and the computed Fourier transform of the distribution to be approximated. The minimisation is a typical interpolation problem and can be solved by various numerical methods [PTVF92]. We use a simulated annealing approach that minimises the difference of only a few most

significant harmonics of the Fourier transforms, which is very fast if provided with a good initial solution. We choose the initial solution in such way that the first moment of the real and approximated distribution coincide.

By replacing all transitions whose firing delays have generalised probability distributions (as shown in Figure 5.5(a)) with subnets of the type depicted in Figure 5.5(b) we obtain a CTMC that approximates the non-Markovian marking process of the CGPN. It is obvious that the introduced additional places trigger a potentially huge increase in the size of the TRG and implicitly in the size of the resulted CTMC. The next section details how to efficiently handle such an increase in the dimensions of the underlying stochastic process.

5.6 Approximating Markov Chain Construction

The marking process of a Petri Net that models an application under the assumptions stated in Section 5.1, such as the one depicted in Figure 5.3, is a generalised semi-Markov process. If we replace the transitions that have generally distributed firing delay probabilities with Coxian distributions, the resulting Petri Net has a marking process that is a continuous time Markov chain that approximates the generalised semi-Markov process. In this section we show how to express the infinitesimal generator of the CTMC by means of generalised tensor sums and products.

Plateau and Fourneau [PF91] have shown that the infinitesimal generator of a CTMC underlying a parallel system can be expressed as a generalised tensor sum [Dav81] of the local infinitesimal generators, i.e. the local generators of the parallel subsystems. Haddad et al. [HMC97] analysed marking processes of Generalised Stochastic Petri Nets with Coxian and phase-type distributions. They classify the transitions of the net in local and non-local transitions and use the results of Plateau and Fourneau. However, they place certain restrictions on the usage of immediate transitions. Moreover, because under our assumptions the firing of the $Clock$ transition (see Figure 5.3) may disable all the other timed transitions, there exists no partition of the set of places such that the results of Plateau and Fourneau, and Haddad might apply.

In our case, the infinitesimal generator is not a sum of tensor products and sums. Rather, the infinitesimal generator matrix is partitioned into submatrices, each of them being a sum of tensor expressions.

Let S be the set of states of the GSMP underlying the Petri Net before the replacement outlined in the previous section. This GSMP corresponds to the TRG of the Petri Net model. Let $\mathcal{M} = [m_{ij}]$ be a square matrix of size $|S| \times |S|$ where $m_{ij} = 1$ if there exists a transition from the state s_i to the state s_j in the GSMP and $m_{ij} = 0$ otherwise. We first partition the set of states S in clusters such that states in the same cluster have outgoing edges labelled with the same set of transitions. A cluster is identified by a binary combination that indicates the set of transitions that are enabled in the particular cluster (which, implicitly, also indicates the set of tasks that are running in the states belonging

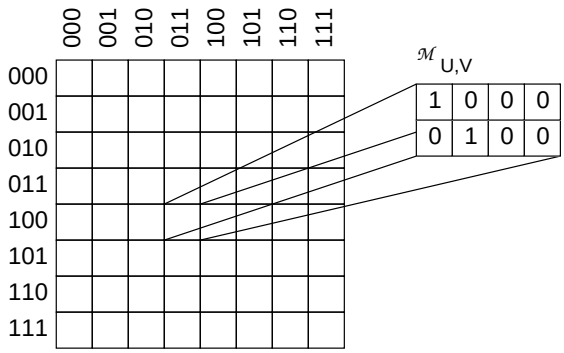


Figure 5.6: The matrix corresponding to a GSMP

to that particular cluster). The clusters are sorted according to their corresponding binary combination and the states in the same cluster are consecutively numbered.

Consider an application with three independent tasks, each of them mapped on a different processor. In this case, 8 clusters can be formed, each corresponding to a possible combination of simultaneously running tasks. Note that if the tasks were not independent, the number of combinations of simultaneously running tasks, and implicitly of clusters, would be smaller. The cluster labelled with 101, for example, contains states in which the tasks τ_1 and τ_3 are running.

Figure 5.6 depicts the matrix $\mathcal{M}$ corresponding to the GSMP of the application described above. The rows and columns in the figure do not correspond to individual rows and columns in $\mathcal{M}$. Each row and column in Figure 5.6 corresponds to one *cluster* of states. The row labelled with 100, for example, as well as the column labelled with the same binary number, indicate that the task τ_1 is running in the states belonging to the cluster labelled with 100, while tasks τ_2 and τ_3 are not running in the states belonging to this cluster. Each cell in the figure does not correspond to a matrix element but to a submatrix $\mathcal{M}_{l_i,l_j}$, where $\mathcal{M}_{l_i,l_j}$ is the incidence matrix corresponding to the clusters labelled with l_i and l_j (an element of $\mathcal{M}_{l_i,l_j}$ is 1 if there is a transition from the state corresponding to its row to the state corresponding to its column, and it is 0 otherwise). The submatrix $\mathcal{M}_{U,V}$ at the intersection of the row labelled with $U = 100$ and the column labelled with $V = 011$ is detailed in the figure. The cluster labelled with $U = 100$ contains 2 states (corresponding to the two rows of the magnified cell in Figure 5.6), while the cluster labelled with $V = 011$ contains 4 states (corresponding to the four columns of the magnified cell in Figure 5.6). As shown in the figure, when a transition from the first state of the cluster labelled with U occurs, the first state of the cluster labelled with V is reached (corresponding to the 1 in the intersection of the first row and first column in the magnified cell). This corresponds to the case when τ_1 completes execution (τ_1 is the only running task in the states belonging to the cluster labelled with U) and τ_2 and τ_3 are subsequently started (τ_2 and τ_3 are the running tasks in the states belonging to the cluster labelled with V).

Once we have the matrix $\mathcal{M}$ corresponding to the underlying GSMP, the next step is the generation of the CTMC using the Coxian distribution for approximation of arbitrary probability distributions of transi-

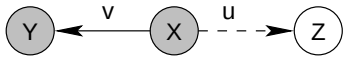


Figure 5.7: Part of a GSMP

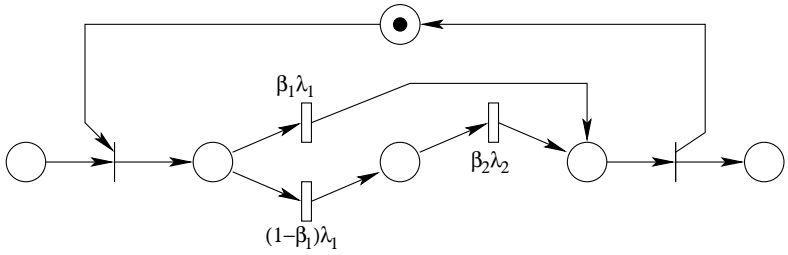


Figure 5.8: Coxian approximation with two stages

tion delays. When using the Coxian approximation, a set of new states is introduced for each state in S (S is the set of states in the GSMP), resulting in an expanded state space S', the state space of the approximating CTMC. We have to construct a matrix $\mathcal{Q}$ of size $|S'| \times |S'|$, the so called infinitesimal generator of the approximating CTMC. The construction of $\mathcal{Q}$ is done cell-wise: for each submatrix of $\mathcal{M}$, a corresponding submatrix of $\mathcal{Q}$ will be generated. Furthermore, null submatrices of $\mathcal{M}$ will result in null submatrices of $\mathcal{Q}$. A cell $\mathcal{Q}_{U,V}$ of $\mathcal{Q}$ will be of size $G \times H$, where

$$G = |U| \cdot \prod_{i \in EnU} r_i$$

$$H = |V| \cdot \prod_{i \in EnV} r_i$$

and U and V are clusters of states, $|U|$ and $|V|$ denote the number of states belonging to the respective clusters, $EnU = \{k : \text{transition } e_k$, corresponding to the execution of task τ_k, is enabled in $U\}$, $EnV = \{k :$ transition e_k, corresponding to the execution of task τ_k, is enabled in $V\}$, and r_k is the number of stages we use in the Coxian approximation of the ETPDF of task τ_k.

We will illustrate the construction of a cell in $\mathcal{Q}$ from a cell in $\mathcal{M}$ using an example. We consider a cell on the main diagonal, as it is the most complex case. Let us consider three states in the GSMP depicted in Figure 5.7. Two tasks, τ_u and τ_v, are running in the states X and Y. These two states belong to the same cluster, labelled with 11. Only task τ_v is running in state Z. State Z belongs to cluster labelled with 10. If task τ_v finishes running in state X, a transition to state Y occurs in the GSMP. This corresponds to the situation when a new instantiation of τ_v becomes active immediately after the completion of a previous one. When task τ_u finishes running in state X, a transition to state Z occurs in the GSMP. This corresponds to the situation when a new instantiation of τ_u is not immediately activated after the completion of a previous one. Consider that the probability distribution of the execution time of task τ_v is approximated with the three stage Coxian distribution depicted in Figure 5.5(b) and that of τ_u is approximated with the two stage Coxian distribution depicted in Figure 5.8. The resulting CTMC corresponding to the part of the GSMP in Figure 5.7 is depicted in Figure 5.9. The edges between the states are labelled with

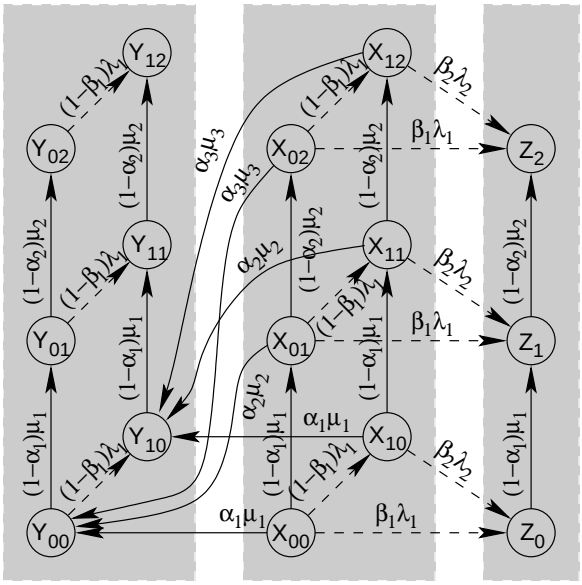


Figure 5.9: Expanded Markov chain

the average firing rates of the transitions of the Coxian distributions. Dashed edges denote state changes in the CTMC caused by firing of transitions belonging to the Coxian approximation of the ETPDF of τ_u. Solid edges denote state changes in the CTMC caused by firing of transitions belonging to the Coxian approximation of the ETPDF of τ_v. The state Y_{12}, for example, corresponds to the situation when the GSMP in Figure 5.7 would be in state Y and the first two of the three stages of the Coxian distribution approximating the ETPDF of τ_v (Figure 5.5(b)) and the first stage out of the two of the Coxian distribution approximating the ETPDF of τ_u (Figure 5.8) have fired.

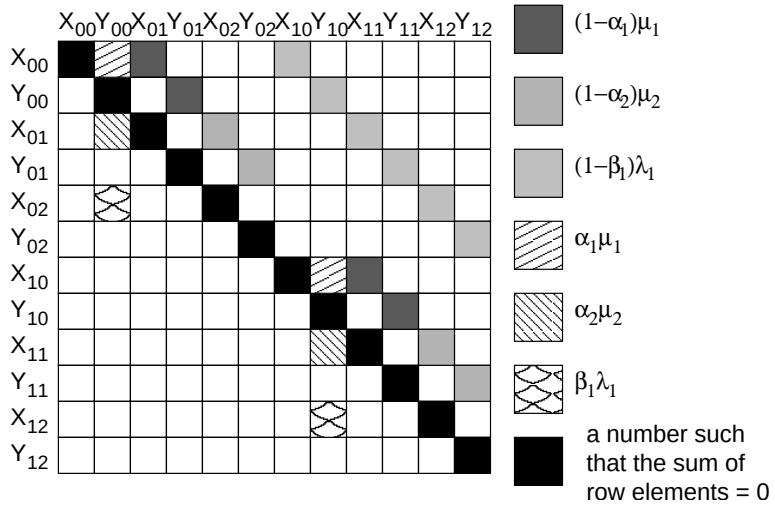


Figure 5.10: The cell $\mathcal{Q}_{(11),(11)}$ corresponding to the example in Figure 5.9

Let us construct the cell $\mathcal{Q}_{(11),(11)}$ on the main diagonal of the infinitesimal generator $\mathcal{Q}$ of the CTMC in Figure 5.9. The cell is situated at the intersection of the row and column corresponding to the cluster labelled with 11 and is depicted in Figure 5.10. The matrix $\mathcal{Q}_{(11),(11)}$ contains the average transition rates between the states X_{ij} and Y_{ij}, $0 \leq i \leq 1$, $0 \leq j \leq 2$, of the CTMC in Figure 5.9 (no state Z, only states X and Y belong to the cluster labelled with 11). The observed regularity in the structure of the stochastic process in Figure 5.9 is reflected in the expression of $\mathcal{Q}_{(11),(11)}$ as shown in Figure 5.10. Because $\mathcal{Q}$ is a generator matrix (sum of row elements equals 0), some negative elements have to be introduced on the main diagonal that do not correspond to transitions in the chain depicted in Figure 5.9. The expression of $\mathcal{Q}_{(11),(11)}$ is given below:

$$\mathcal{Q}_{(11),(11)} = (\mathcal{A}_u \oplus \mathcal{A}_v) \otimes I_{|11|} + I_{r_u} \otimes \mathcal{B}_v \otimes e_{r_v} \otimes \mathcal{D}_v$$

where

$$\mathcal{A}_u = \begin{bmatrix} -\lambda_1 & (1-\beta_1)\lambda_1 \\ 0 & -\lambda_2 \end{bmatrix} \qquad \mathcal{B}_v = \begin{bmatrix} \alpha_1\mu_1 \\ \alpha_2\mu_2 \\ \alpha_3\mu_3 \end{bmatrix} \qquad \mathcal{D}_v = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{5.1}$$

and

$$\mathcal{A}_v = \begin{bmatrix} -\mu_1 & (1-\alpha_1)\mu_1 & 0 \\ 0 & -\mu_2 & (1-\alpha_2)\mu_2 \\ 0 & 0 & -\mu_3 \end{bmatrix} \qquad e_{r_v} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \tag{5.2}$$

$|11|$ denotes the size of the cluster labelled with 11. I_i is the identity matrix of size $i \times i$, r_i indicates the number of stages of the Coxian distribution that approximates the ETPDF of task τ_i. $\oplus$ and $\otimes$ are the Kronecker sum and product of matrices, respectively.

In general, a matrix $\mathcal{A}_k = [a_{ij}]$ is an $r_k \times r_k$ matrix, and is defined as follows:

$$a_{ij} = \begin{cases} (1-\alpha_{ki})\mu_{ki} & \text{if } j = i+1 \\ -\mu_{ki} & \text{if } j = i \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

where α_{ki} and μ_{ki} characterise the i^{th} stage of the Coxian distribution approximating a transition t_k.

A matrix $\mathcal{B}_k = [b_{ij}]$ is an $r_k \times 1$ matrix and $b_{i1} = \alpha_{ki} \cdot \mu_{ki}$. A matrix $e_{rk} = [e_{ij}]$ is a $1 \times r_k$ matrix and $e_{11} = 1$, $e_{1i} = 0, 1 < i \leq r_k$. A matrix $\mathcal{D}_k = [d_{ij}]$ corresponding to a cell U, V is a $|U| \times |V|$ matrix defined as follows:

$$d_{ij} = \begin{cases} 1 & \text{if an edge labelled with } k \text{ links} \\ & \text{the } i^{th} \text{ state of } U \text{ with the } j^{th} \text{ state of } V \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

In general, considering a label U, the cell $\mathcal{Q}_{U,U}$ on the main diagonal of $\mathcal{Q}$ is obtained as follows:

$$\begin{aligned} \mathcal{Q}_{U,U} = & \left(\bigoplus_{i \in EnU} \mathcal{A}_i \right) \otimes I_{|U|} + \\ & + \sum_{i \in EnU} \left(\bigotimes_{\substack{j \in EnU \\ j > i}} I_{r_j} \right) \otimes \mathcal{B}_i \otimes e_{r_i} \otimes \left(\bigotimes_{\substack{j \in EnU \\ j < i}} I_{r_j} \right) \otimes \mathcal{D}_i \end{aligned} \tag{5.5}$$

A cell situated at the intersection of the row corresponding to label U with the column corresponding to label V ($U \neq V$) is obtained as follows:

$$\mathcal{Q}_{U,V} = \sum_{i \in EnU} \left(\bigotimes_{j \in EnU \cup EnV} \mathcal{F}_{ij} \right) \otimes \mathcal{D}_i \tag{5.6}$$

The matrices $\mathcal{F}$ are given by the following expression:

$$\mathcal{F}_{ij} = \begin{cases} v_{r_j} & \text{if } j \in EnU \wedge j \notin EnV \wedge j \neq i \\ I_{r_j} & \text{if } j \in EnU \wedge j \in EnV \wedge j \neq i \\ \mathcal{B}_i & \text{if } j \notin EnV \wedge j = i \\ \mathcal{B}_i \otimes e_{r_i} & \text{if } j \in EnV \wedge j = i \\ e_{r_j} & \text{if } j \notin EnU \end{cases} \tag{5.7}$$

where $v_{r_k} = [v_{i1}]$ is a $r_k \times 1$ matrix, $v_{i1} = 1$, $1 \leq i \leq r_k$.

The solution of the CTMC implies solving for π in the following equation:

$$^t\pi \cdot \mathcal{Q} = 0 \tag{5.8}$$

where π is the steady-state probability vector (and $^t\pi$ is its transpose) and $\mathcal{Q}$ the infinitesimal generator of the CTMC.

We conclude this section with a discussion on the size of $\mathcal{Q}$ and its implications on analysis time and memory. The submatrix $\mathcal{Q}_{U,U}$, defined by Eq.(5.5), has $(|EnU| + 1 - \sum_{i \in EnU} \frac{1}{r_i}) \cdot \prod_{i \in EnU} r_i \cdot |U|$ non-zero elements. Similarly, for a given cluster label U, all the submatrices $\mathcal{Q}_{U,V}, U \neq V$, have approximately $|EnU| \cdot \prod_{i \in EnU} r_i \cdot |U|$ non-zero elements on aggregate (see Eq.(5.6)). Letting $Z_U = \prod_{i \in EnU} r_i \cdot |U|$, the entire generator $\mathcal{Q}$ has approximately

$$|\mathcal{Q}| \approx \sum_U (2 \cdot |EnU| + 1 - \sum_{i \in EnU} \frac{1}{r_i}) \cdot Z_U \tag{5.9}$$

non-zero elements. The state probability vector π has

$$|\pi| = \sum_U Z_U = \sum_U |U| \cdot \prod_{i \in EnU} r_i \tag{5.10}$$

elements.

Let us suppose that we store the matrix $\mathcal{Q}$ in memory. Then, in order to solve the equation $^t\pi \cdot \mathcal{Q} = 0$, we would need

$$|\pi| + \xi \cdot |\mathcal{Q}| = \sum_U Z_U + \xi \cdot \sum_U (2 \cdot |EnU| + 1 - \sum_{i \in EnU} \frac{1}{r_i}) \cdot Z_U \tag{5.11}$$

memory locations, where ξ is the number of information items characterising each non-zero element of the matrix. ξ reflects also the overhead of sparse matrix element storage.

As can be seen from the expressions of $\mathcal{Q}_{U,U}$ and $\mathcal{Q}_{U,V}$, the matrix $\mathcal{Q}$ is completely specified by means of the matrices $\mathcal{A}_i$, $\mathcal{B}_i$, and $\mathcal{D}_i$ (see Eq.(5.5) and (5.6)), hence it needs not be stored explicitly in memory, but its elements are generated on-the-fly during the numerical solving of the CTMC. In this case, we would need to store

$$\begin{aligned} |\pi| + \sum_i |\mathcal{A}_i| + \sum_i |\mathcal{B}_i| + \sum_i |\mathcal{D}_i| = \sum_U Z_U + \\ + \sum_i (3 \cdot r_i - 1) + \sum_U \xi \cdot |EnU| \cdot |U| \end{aligned} \tag{5.12}$$

values in order to solve ${}^t\pi \cdot \mathcal{Q} = 0$. Even for large applications, the matrices $\mathcal{A}_i$ and $\mathcal{B}_i$ are of negligible size ($\sum_i (2 \cdot r_i - 1)$ and $\sum_i r_i$ respectively). The ratio between the memory space needed by $\mathcal{Q}$ if stored and if generated on-the-fly is

$$\frac{\sum_U Z_U + \xi \cdot \sum_U (2 \cdot |EnU| + 1 - \sum_{i \in EnU} \frac{1}{r_i}) Z_U}{\sum_U Z_U + \sum_i (3 \cdot r_i - 1) + \xi \cdot \sum_U |EnU| \cdot |U|} \tag{5.13}$$

For the application example in Figure 5.2, the expression in Eq.(5.13) evaluates to 11.12 if Coxian distributions with 6 stages substitute the original distributions. The actual memory saving factor, as indicated by our analysis tool, is 9.70. The theoretical overestimation is due to the fact that possible overlaps of non-null elements of matrices $\mathcal{Q}_{UV}$ were not taken into account in the Eq.(5.11). Nevertheless, we see that even for small applications memory savings of one order of magnitude can be achieved by exploiting the special structure of the infinitesimal generator of the approximating CTMC. Further evaluations will be presented in Section 5.8.

The drawback of this approach is the time overhead in order to generate each non-zero element of $\mathcal{Q}$ on-the-fly. A naïve approach would need $O(|EnU| \cdot |EnV|)$ arithmetic operations in order to compute each non-zero element of the cell (U, V). In the worst case, the number of arithmetic operations is $O(M^2)$, where M is the number of processors of the system. However, a large body of research [BCKD00] provides intelligent numerical algorithms for matrix-vector computation that exploit factorisation, reuse of temporary values, and reordering of computation steps. Thus, the aforementioned overhead is significantly amortised.

5.7 Extraction of Results

This section describes how the deadline miss ratios for each task and task graph are calculated, once the CTMC approximating the stochastic process underlying the system is solved.

As described in Section 5.3, the event of a task (task graph) missing its deadline corresponds to identifiable edges in the tangible reachability graph of the CGPN modelling the application (and implicitly to transitions in the underlying GSMP). The expected deadline miss ratio of a task (task graph) can be computed as the sum of the expected transition rates of the corresponding identified edges multiplied with the task (task graph) period.

Not only deadline miss events may be modelled as transitions along certain edges in the TRG, but also more complex events. Such events are, for example, the event that a task graph misses its deadline when a certain task τ_i missed its deadline. Another example is the event that task τ_j missed its deadline when task τ_i started later than a given time moment. There is indeed a large number of events that may be represented as transitions along identifiable edges in the TRG. Inspecting such events can be extremely useful for diagnosis, finding performance bottlenecks and non-obvious correlations between deadline miss events, detecting which task needs to be re-implemented, or whose execution is badly scheduled such that other processors idle. Such kind of information can be exploited for the optimisation of the application.

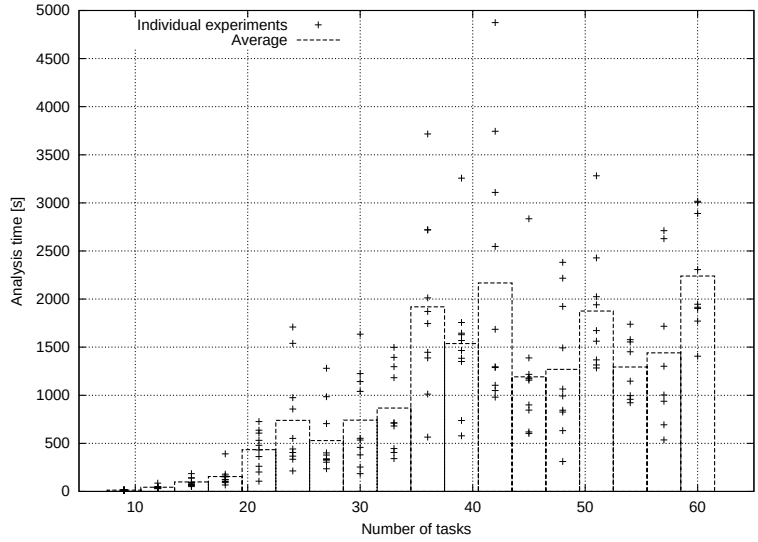


Figure 5.11: Analysis time vs. number of tasks

In conclusion, once the steady state probabilities of the stochastic process are obtained, we are interested in the expected rate of certain edges in the TRG of the CGPN. We illustrate how to calculate this rate based on an example.

Let us consider the edge $X \rightarrow Z$ in the GSMP in Figure 5.7. The edges $X_{00} \rightarrow Z_0$, $X_{10} \rightarrow Z_0$, $X_{01} \rightarrow Z_1$, $X_{11} \rightarrow Z_1$, $X_{02} \rightarrow Z_2$, and $X_{12} \rightarrow Z_2$ in the CTMC in Figure 5.9, which approximates the GSMP in Figure 5.7, correspond to the edge $X \rightarrow Z$ in the GSMP. The expected transition rate of $X \rightarrow Z$ can be approximated by means of the expected transition rates of the corresponding edges in the CTMC and is given by the expression

$$(\pi_{X_{00}} + \pi_{X_{01}} + \pi_{X_{02}}) \cdot \beta_1\lambda_1 + (\pi_{X_{10}} + \pi_{X_{11}} + \pi_{X_{12}}) \cdot \beta_2\lambda_2 \qquad (5.14)$$

where β_1, β_2, λ_1, and λ_2 characterise the Coxian distribution that approximates the probability distribution of the delay of the transition $X \rightarrow Z$ (in this case, the ETPDF of τ_v). $\pi_{X_{ij}}$ is the probability of the CTMC being in state X_{ij} after the steady state is reached. The probabilities $\pi_{X_{ij}}$ are obtained as the result of the numerical solution of the CTMC (Eq(5.8)).

5.8 Experimental Results

We performed four sets of experiments. All were run on an AMD Athlon at 1533 MHz.

5.8.1 Analysis Time as a Function of the Number of Tasks

The first set of experiments investigates the dependency of the analysis time on the number of tasks in the system. Sets of random task graphs were generated, with 9 to 60 tasks per set. Ten different sets were generated and analysed for each number of tasks per set. The underlying

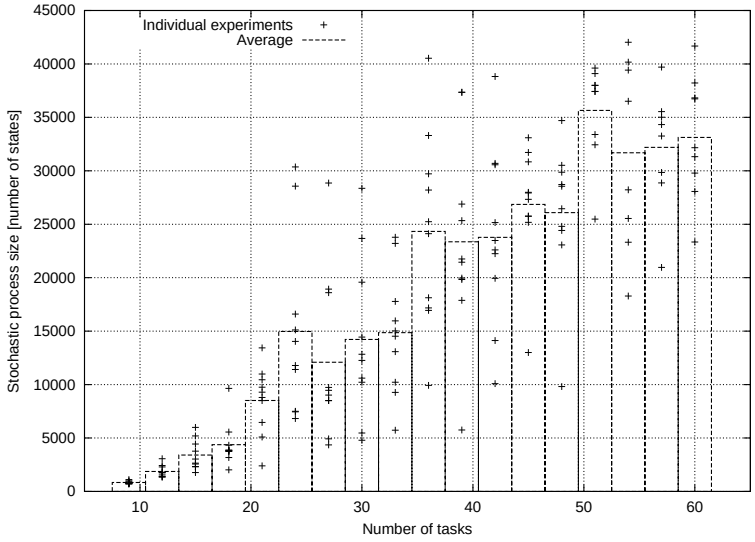


Figure 5.12: Stochastic process size vs. number of tasks

architecture consists of two processors. The task execution time probability distributions were approximated with Coxian distributions with 2 to 6 stages. The dependency between the needed analysis time and the number of tasks is depicted in Figure 5.11. The crosses indicate the analysis times of the individual applications, while the boxes represent the average analysis time for classes of application with the same number of tasks. The analysis time depends on the size of the stochastic process to be analysed (in terms of number of states) as well as on the convergence rate of the numerical solution of the CTMC. This variation of the convergence rate of the numerical solution does not allow us to isolate the effect of the number of tasks on the analysis time. Therefore, in Figure 5.12 we depicted the influence of the number of tasks on the size of the CTMC in terms of number of states. As seen from the figure, the number of states of the CTMC increases following a linear tendency with the number of tasks. The observed slight non-monotonicity stems from other parameters that influence the stochastic process size, such as task graph periods, and the amount of task parallelism.

5.8.2 Analysis Time as a Function of the Number of Processors

In the second set of experiments, we investigated the dependency between the analysis time and the number of processors. Ten different sets of random task graphs were generated. For each of the ten sets, 5 experiments were performed, by allocating the 18 tasks of the task graphs to 2 to 6 processors. The results are plotted in Figure 5.13. It can be seen that the analysis time is exponential in the number of processors. The exponential character is induced by the possible task execution orderings. The number of task execution orderings increases exponentially with increased parallelism provided by the architecture. We see from Eq.(5.10) that the number of states of the CTMC is exponentially dependent on $|EnU|$, the number of simultaneously enabled transitions in the states of cluster $|U|$. $|EnU|$ is at most $M + 1$, where

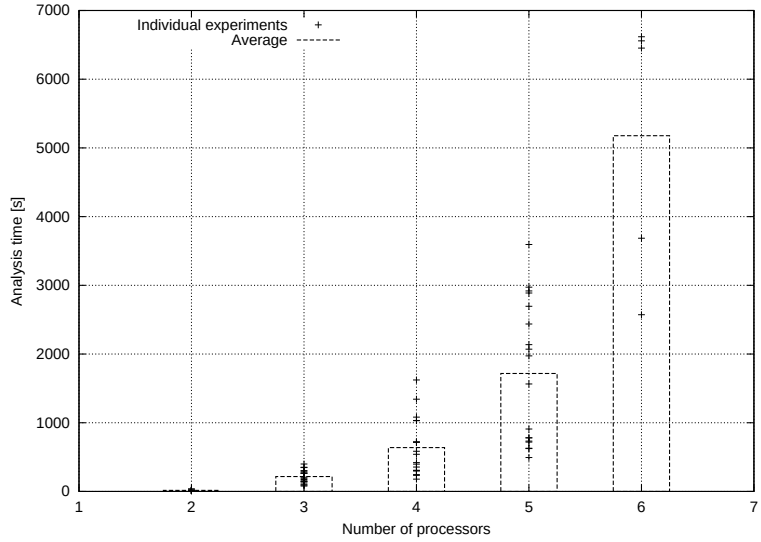


Figure 5.13: Analysis time vs. number of processors

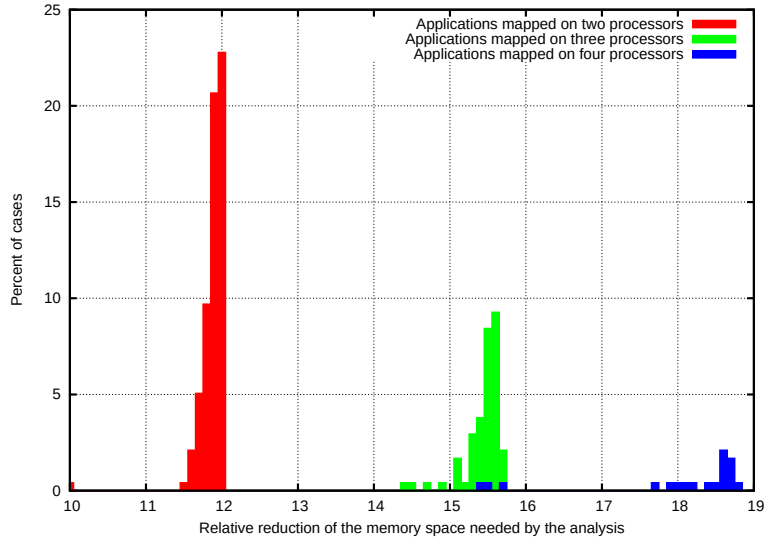


Figure 5.14: Histogram of memory reduction

M is the number of processes. Thus, the experiment confirms the theoretical result that the number of states of the CTMC is exponential in the number of processors.

5.8.3 Memory Reduction as a Consequence of the On-the-Fly Construction of the Markov Chain Underlying the System

In the third set of experiments, we investigated the reduction in the memory needed in order to perform the CTMC analysis when using on-the-fly construction of the infinitesimal generator based on equations (5.5) and (5.6). We constructed 450 sets of synthetic applications

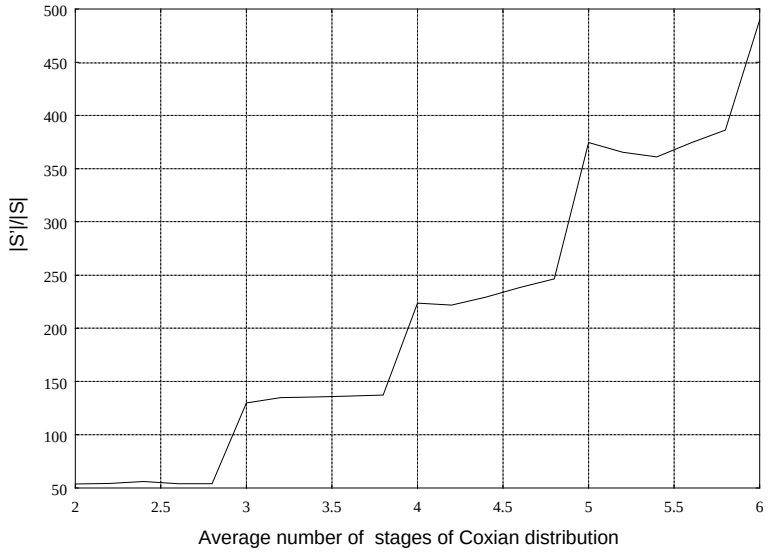


Figure 5.15: Increase in stochastic process size with number of stages for approximating the arbitrary ETPDFs

with 20 to 40 tasks each. The execution time probability distributions were approximated with Coxian distributions with 6 stages. For each application, we ran our analysis twice. In the first run, the entire infinitesimal generator of the CTMC approximating the stochastic process underlying the application was stored in memory. In the second run, the elements of the infinitesimal generator were computed on-demand during the analysis and not stored. In both runs, we measured the memory needed for the analysis, and calculated the relative memory reduction in the case of the on-demand generation. The histogram of the relative memory reduction is shown in Figure 5.14. We observe a memory reduction ranging from 10 to 19 times. Additionally, we observe a strong correlation between the memory reduction factor and the number of processors on which the application is mapped. Also, the memory reduction factor increases with the number of processors. Last, we observe that only 48% of the applications mapped on three processors and only 13.3% of the applications mapped on four processors could have been analysed in both runs, such that the comparison can be made. For 52% and 86.7% of applications mapped on 3 respectively 4 processors, the 512MB memory of a desktop PC computer were insufficient for the analysis in the case the entire infinitesimal generator is stored in memory. The cases in which the comparison could not be made were not included in the histogram.

5.8.4 Stochastic Process Size as a Function of the Number of Stages of the Coxian Distributions

In the fourth set of experiments, we investigated the increase in the stochastic process size induced by using different number of stages for approximating the arbitrary ETPDFs. We constructed 98 sets of random task graphs ranging from 10 to 50 tasks mapped on 2 to 4 processors. The ETPDFs were approximated with Coxian distributions using 2 to 6 stages. The results for each type of approximation were averaged

Table 5.1: Accuracy vs. number of stages

	2 stages	3 stages	4 stages	5 stages
Relative error	8.467%	3.518%	1.071%	0.4%

over the 98 sets of graphs and the results are plotted in Figure 5.15. Recall that $|S|$ is the size of the GSMP and $|S'|$ is the much larger size of the CTMC obtained after approximation. As more stages are used for approximation, as larger the CTMC becomes compared to the original GSMP. As shown in Section 5.6, in the worst case, the growth factor is

$$\prod_{i \in E} r_i \tag{5.15}$$

As can be seen from Figure 5.15, the real growth factor is smaller than the theoretical upper bound. It is important to emphasise that the matrix $\mathcal{Q}$ corresponding to the CTMC does not need to be stored, but only a vector with the length corresponding to a column of $\mathcal{Q}$. The growth of the vector length with the number of Coxian stages used for approximation can be easily derived from Figure 5.15. The same is the case with the growth of analysis time, which follows that of the CTMC.

5.8.5 Accuracy of the Analysis as a Function of the Number of Stages of the Coxian Distributions

The fifth set of experiments investigates the accuracy of results as a factor of the number of stages used for approximation. This is an important aspect in deciding on a proper trade-off between quality of the analysis and cost in terms of time and memory. For comparison, we used analysis results obtained with our approach elaborated in the previous chapter. That approach is an exact one based on solving the underlying GSMP. However, because of complexity reasons, it can efficiently handle only monoprocessor systems. Therefore, we applied the approach presented in this chapter to a monoprocessor example, which has been analysed in four variants using approximations with 2, 3, 4, and 5 stages. The relative error between missed deadline ratios resulted from the analysis using the approximate CTMC and the ones obtained from the exact solution is presented in Table 5.1. The generalised ETPDFs used in this experiment were created by drawing Bézier curves that interpolated randomly generated control points. It can be observed that good quality results can already be obtained with a relatively small number of approximation stages.

5.8.6 Encoding of a GSM Dedicated Signalling Channel

Finally, we considered the telecommunication application described in Section 4.3.6, namely the baseband processing of a stand-alone dedicated control channel of the GSM. In Section 4.3.6, the application was mapped on a single processor. This implementation could be inefficient as it combines the signal processing of the FIRE encoding and convolutional encoding, and the bit-operation-intensive interleaver and ciphering on one hand with the control dominant processing of the modulator

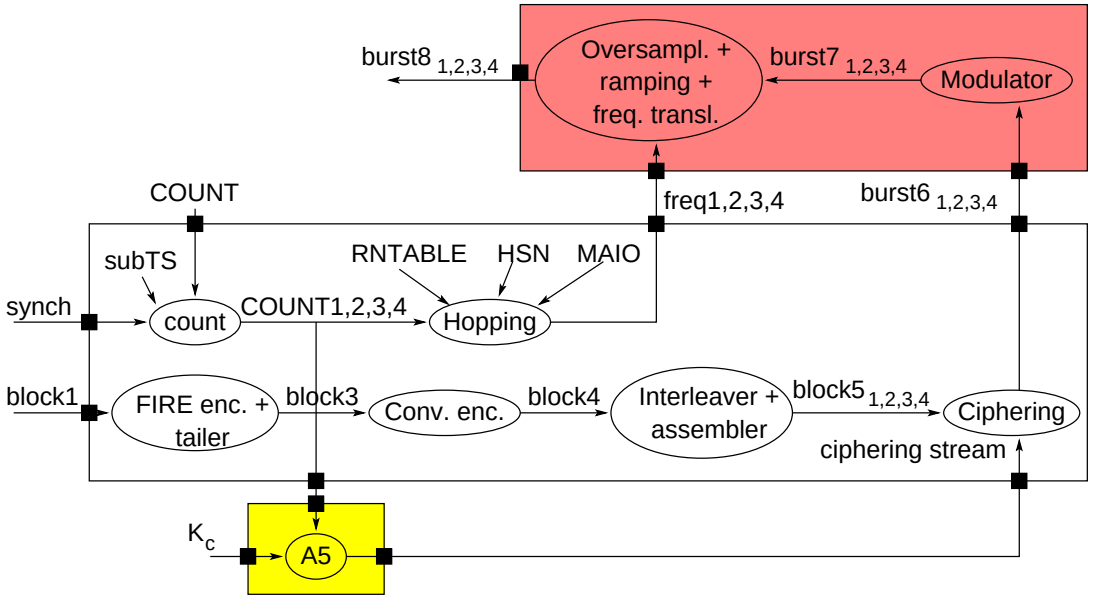


Figure 5.16: Encoding and mapping of a GSM dedicated signalling channel

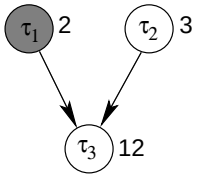


Figure 5.17: Application example

on the other hand. Moreover, the implementation of the publicly unavailable A5 algorithm could be provided as a separate circuit. Thus, in this experiment we consider a mapping as shown in Figure 5.16. In the implementation alternative depicted in the figure, the FIRE and convolutional encodings, the bit interleaving and ciphering, as well as the hopping and count tasks are mapped on a digital signal processor. The A5 task is executed by a ASIC, while the modulating and oversampling tasks are mapped on a different ASIC.

In the case of the 9 tasks depicted in Figure 5.16, the analysis reported an acceptable miss deadline ratio after an analysis time of 4.8 seconds. The ETPDFs were approximated by Coxian distributions with 6 stages. If we attempt to perform the baseband processing of an additional channel on the same DSP processor, three more tasks, namely an additional FIRE and convolutional encoding, and interleaving tasks, are added to the task graph. The analysis in this case took 5.6 seconds. As a result of the analysis, in the case of two channels (9 tasks in total), 10.05% of the deadlines are missed, which is unacceptable according to the application specification.

5.9 Extensions

Possible extensions that address the three restrictions that we assumed on the system model (Section 5.1.3) are discussed in this section.

5.9.1 Individual Task Periods

As presented in Section 5.1.3, we considered that all the tasks belonging to a task graph have the same period. This assumption can be relaxed as follows. Each task $\tau_i \in \Gamma_j$ has its own period π_{τ_i}, with the restriction that π_{τ_i} is a common multiple of all periods of the tasks in ${}^\circ\tau_i$ (π_{τ_i} is an integer multiple of π_{τ_k}, where $\tau_k \in {}^\circ\tau_i$). In this case, π_{Γ_j}, the period of the task graph Γ_j, is equal to the least common multiple of all π_{τ_i}, where π_{τ_i} is the period of τ_i and $\tau_i \in V_j$. The introduction of individual task periods implies the existence of individual task deadlines: each task τ_i has its own deadline $\delta_{\tau_i} = \pi_{\tau_i}$ and the deadline δ_{Γ_j} of a task graph Γ_j (the time by which all tasks $\tau_i \in V_j$ have to have finished) is π_{Γ_j}.

In order to illustrate how applications under such an extension are modelled, let us consider the application example depicted in Figure 5.17: one task graph consisting of the three tasks τ_1, τ_2 and τ_3, where τ_1 is mapped on processor P_2 and τ_2 and τ_3 are mapped on processor P_1. Task τ_1 has period 2, task τ_2 has period 3 and task τ_3 has period 12 as indicated by the numbers near the circles in the figure.

The task graph period is the least common multiple of the periods of the tasks that belong to it, i.e. $\pi_{\Gamma_1} = 12$ for our example.

Figure 5.18 depicts the CGPN that models the application described above. Whenever an instantiation of task τ_1 completes its execution (transition e_1 fires), a token is added in place $r_{3,1}$. Similarly, whenever an instantiation of task τ_2 completes its execution, a token is added in place $r_{3,2}$. In order to be ready to run, an instantiation of task τ_3 needs $\pi_{\tau_3}/\pi_{\tau_1} = 6$ data items produced by 6 instantiations of task τ_1 *and* $\pi_{\tau_3}/\pi_{\tau_2} = 4$ data items produced by 4 instantiations of task τ_2. Therefore, the arcs $r_{3,1} \rightarrow r_3$ and $r_{3,2} \rightarrow r_3$ have multiplicity 6 and 4 respectively.

The firing delay $Tick$ of the $Clock$ transition is not any more the greatest common divisor of the task *graph* periods, 12, but the greatest common divisor of the *task* periods, 1. Every two ticks, f_1 fires and a new token is added to place $task_1$, modelling an arrival of a new instantiation of τ_1. Similarly, every three ticks a new token is added to place $task_2$. In general, the fact that an instantiation of task τ_i has not yet completed its execution is modelled by a marking with the property that at least one of the places $r_{i,k}$, a_i or run_i is marked. Hence, an instantiation of task τ_i misses its deadline if and only if f_i fires in a marking with the above mentioned property.

Every 12 ticks, f_{Γ_1} fires and a new token is added to place $graph_1$, modelling a new instantiation of task graph Γ_1. In general, if Bnd_i contains less than b_i tokens when a new instantiation of Γ_i arrives, then at least one instantiation is active in the system. The event of a task graph Γ_i missing its deadline corresponds to the firing of f_{Γ_i} in a marking in which Bnd_i contains less than b_i tokens.

If Bnd_1 is not marked at that time, it means that there are already b_1 active instantiations of Γ_1 in the system. In this case, $replace_1$ fires and the oldest instantiation is removed from the system. Otherwise, $absorb_1$ fires, consuming a token from Bnd_1. This token is added back when $synch_1$ fires modelling the completion of the task graph Γ_1. An instantiation of Γ_1 completes its execution when $\pi_{\Gamma_1}/\pi_{\tau_1} = 6$ instantiations of τ_1, $\pi_{\Gamma_1}/\pi_{\tau_2} = 4$ instantiations of τ_2 and $\pi_{\Gamma_1}/\pi_{\tau_3} = 1$ instantiation of τ_3 complete their execution. Therefore, the arcs $done_1 \rightarrow synch_1$, $done_2 \rightarrow synch_1$, and $done_3 \rightarrow synch_1$, have multiplicity 6, 4 and 1 respectively.

The following experiment has been carried out in order to assess the impact of individual task periods on the analysis complexity. Three sets of test data, $Normal$, $High$, and Low, have been created. The test data in set $Normal$ contains 300 sets of random task graphs, each set comprising 12 to 27 tasks grouped in 3 to 9 task graphs. The tasks are mapped on 2 to 6 processors. Each task has its own period, as described in this section. For any task graph Γ_i, the least common multiple LCM_i and the greatest common divisor, GCD_i, of the periods of the tasks belonging to the task graph are computed. The test data $High$ and Low is identical to the test data $Normal$ with the exception of task periods that are equal for all tasks belonging to the same task graph. All tasks belonging to the task graph Γ_i in the test data $High$ have period LCM_i while the same tasks have period GCD_i in the test data Low.

Figure 5.19 plots the dependency of the average size of the underlying generalised semi-Markov process (number of states) on the number of tasks. The three curves correspond to the three test data sets. The following conclusions can be drawn from the figure:

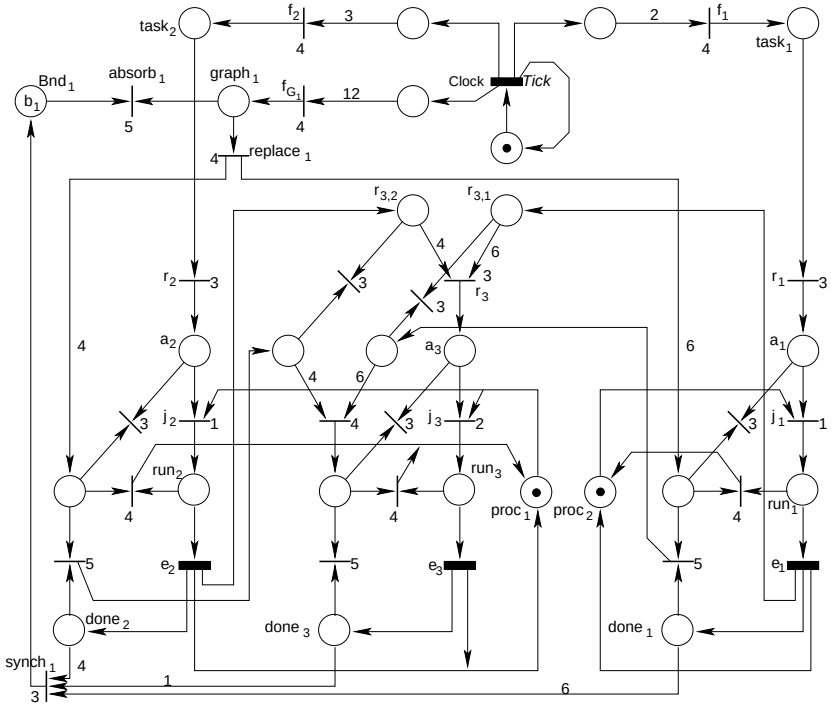


Figure 5.18: CGPN modelling the application in Figure 5.17

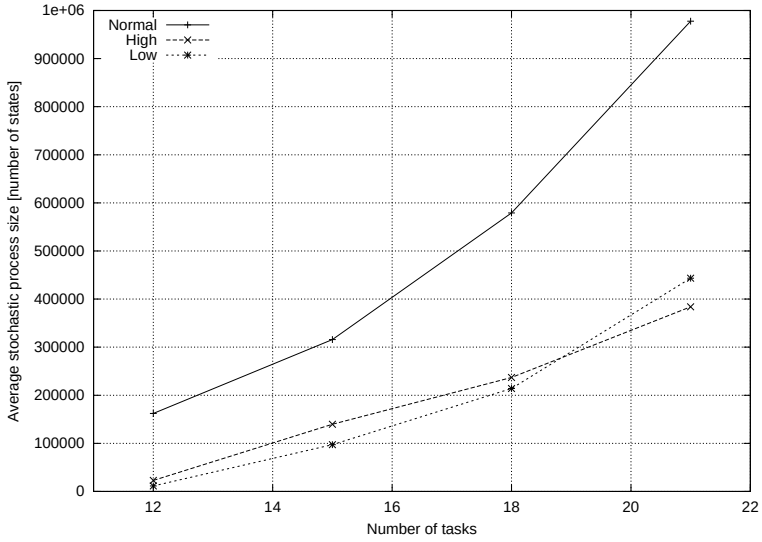


Figure 5.19: Individual task periods compared to uniform task

Table 5.2: Relative increase of the GSMP size in the case of individual periods

	Increase relative to	
Number of tasks	test set Low	test set $High$
12	13.109	6.053
15	2.247	1.260
18	1.705	1.443
21	1.204	1.546

1. The plots corresponding to $High$ and Low are very close to each other. This indicates that the number of states in the GSMP is only weakly dependent on the particular period value.
2. The number of states in the GSMP corresponding to test set $Normal$ is larger than the ones corresponding to test sets $High$ and Low. This confirms that different periods for tasks belonging to the same task graph lead to a relative increase in the GSMP size as compared to the case when all tasks in the same task graph have the same period.
3. The relative growth of the size of the GSMP in the case in which tasks belonging to the same task graph have different periods compared to the case in which tasks belonging to the same task graph have the same period is decreasing with the number of tasks.

Table 5.2 contains the average increase in the size of the generalised semi-Markov processes corresponding to the task sets in test data $Normal$ relative to the size of the generalised semi-Markov processes underlying the task sets in test data Low and $High$.

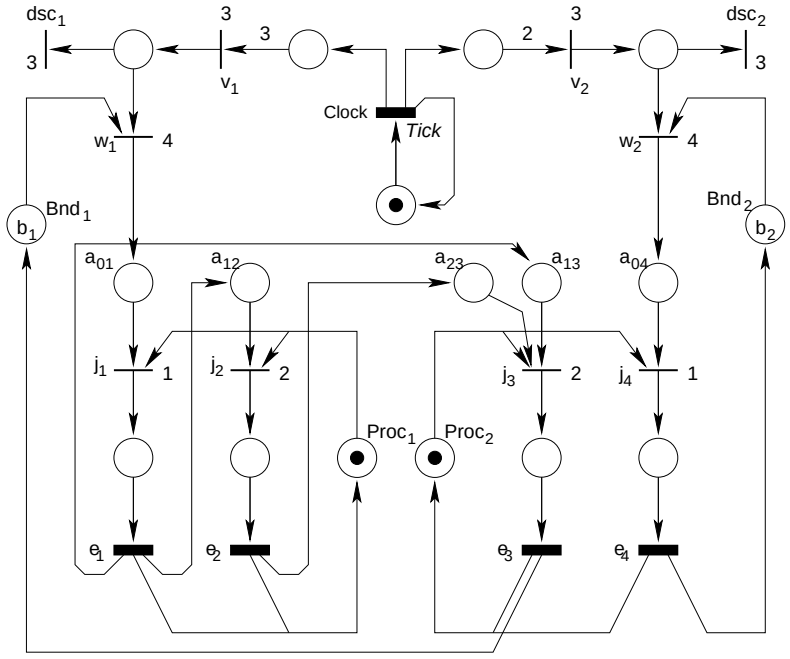


Figure 5.20: CGPN modelling the task graphs in Figure 5.2 in the case of the rejection policy

5.9.2 Task Rejection vs. Discarding

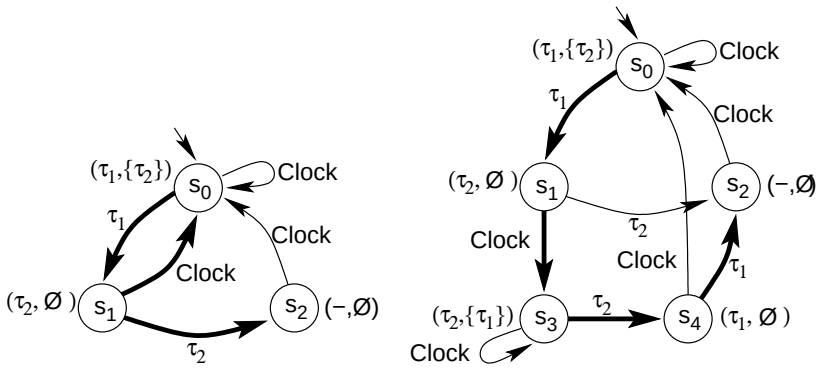
As formulated in Section 5.1.3, when there are b_i concurrently active instantiations of task graph Γ_i in the system, and a new instantiation of Γ_i demands service, the oldest instantiation of Γ_i is eliminated from the system. Sometimes, this behaviour is not desired, as the oldest instantiation might have been very close to finishing, and by discarding it, the invested resources (time, memory, bandwidth, etc.) are wasted, as discussed before.

Therefore, our approach has been extended to support a late task policy in which, instead of discarding the oldest instantiation of Γ_i, the newly arrived instantiation is denied service (rejected) by the system. However, the analysis method supports the rejection policy only in the context of fixed priority scheduling.

The CGPN modelling the application in Figure 5.2, when considering the rejection policy, is depicted in Figure 5.20.

If there are b_i concurrently active instantiations of a task graph Γ_i in the system, then the place Bnd_i contains no tokens. If a new instantiation of Γ_i arrives in such a situation (v_i fires), then dsc_i will fire, "throwing away" the newly arrived instantiation.

Let us suppose that an application consists of two independent tasks, τ_1 and τ_2, they are mapped on different processors and they have the same period π. $b_1 = b_2 = 1$. An instantiation of τ_2 always finishes before the arrival of the next instantiation. Suppose that both tasks are running at time moment t. The marking of the CGPN at time moment t is M_t. The instantiation of τ_1 that is running at time moment t runs an extremely long time, beyond time moment $(\lfloor t/\pi \rfloor + 1)\pi + \epsilon$ ($\epsilon > 0$, but very small). Therefore, the instantiation of τ_1 that arrives at time moment $(\lfloor t/\pi \rfloor + 1)\pi$ is rejected. The marking of the CGPN

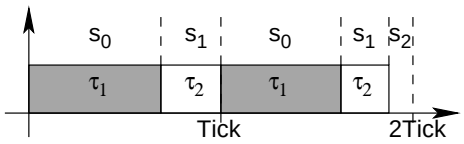


(a) Discarding (b) Rejection

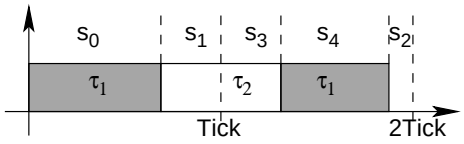
Figure 5.21: Stochastic process underlying the application

$M_{(\lfloor t/\pi \rfloor+1)\pi+\epsilon}$ at time $(\lfloor t/\pi \rfloor + 1)\pi + \epsilon$ is identical to the marking at time moment t, M_t. However, M_t corresponds to the situation when the freshest instantiations of τ_1 and τ_2 are running, while $M_{(\lfloor t/\pi \rfloor+1)\pi+\epsilon}$ corresponds to the situation when an older instantiation of τ_1 and the freshest instantiation of τ_2 are running. Hence, if the task priorities are dynamic, it is impossible to extract their priorities solely from the marking of the CGPN. In the case of discarding, as opposed to rejection, *always* the freshest task instantiations are active in the system. Therefore, their latencies can be computed based on the current time (extracted from the current marking, as shown in Section 5.3.5). Therefore, the rejection policy is supported by our analysis method only in the context of fixed (static) priority scheduling when task priorities are constant and explicit in the CGPN model such that they do not have to be extracted from the net marking. Time-stamping of tokens would be a solution for extending the support of the rejection policy to dynamic priority scheduling. This, however, is expected to lead to a significant increase in the tangible reachability graph of the modelling Petri Net and implicitly in the number of states of the underlying GSMP.

Although CGPNs like the one in Figure 5.20, modelling applications with rejection policy, are simpler than the CGPN in Figure 5.3, modelling applications with discarding policies, the resulting tangible reachability graphs (and implicitly the underlying generalised semi-Markov processes) are larger. In the case of discarding, the task instantiations are always the freshest ones. In the case of rejection, however, the instantiation could be arbitrarily old leading to many more combinations of possible active task instantiations. In order to illustrate this, let us consider the following example. The task set consists of two independent tasks, τ_1 and τ_2 with the same period and mapped on the same processor. Task τ_1 has a higher priority than task τ_2. At most one active instantiation of each task is allowed in the system at each time moment. Figure 5.21(a) depicts the underlying generalised semi-Markov process in the case of the discarding policy, while Figure 5.21(b) depicts the underlying generalised semi-Markov process in the case of the rejection policy. The states are annotated by tuples of the form (a, W) where a is the running task and W is the set of ready tasks.



(a) Discarding



(b) Rejection

Figure 5.22: Gantt diagrams for the highlighted paths in Figure 5.21

Table 5.3: Discarding compared to rejection

Tasks	Average GSMP size		Relative increase of the GSMP size
	Discarding	Rejection	
12	8437.85	18291.23	1.16
15	27815.28	90092.47	2.23
18	24089.19	194300.66	7.06
21	158859.21	816296.36	4.13
24	163593.31	845778.31	4.17
27	223088.90	1182925.81	4.30

The labels τ_1 and τ_2 on arcs indicate the completion of the corresponding tasks, while $Clock$ indicates the arrival of new task instantiations every period. Figures 5.22(a) and 5.22(b) depict the Gantt diagrams corresponding to the two highlighted paths in the stochastic processes depicted in Figure 5.21(a) and 5.21(b). The Gantt diagrams are annotated with the states in the corresponding stochastic processes. As seen, the rejection policy introduces states, like the priority inversion noted in state s_3, which are impossible when applying discarding.

In order to assess the impact of the rejection policy on the analysis complexity compared to the discarding policy, the following experiments were carried out. 109 task sets of 12 to 27 tasks grouped in 2 to 9 task graphs were randomly generated. Each task set has been analysed two times, first considering the discarding policy and then considering the rejection policy. The results were averaged for task sets with the same cardinality and shown in Table 5.3.

5.9.3 Arbitrary Task Deadlines

As discussed in Section 5.3, a deadline miss is modelled by the firing of a transition capturing the deadline arrival in a marking with certain properties. Therefore, when the task deadline is equal to the corresponding task period, and, implicitly, it coincides with the arrival of a new instantiation, such a transition is already available in the CGPN

model. For example, such transitions are v_i in Figure 5.3 and Figure 5.20 and f_i in Figure 5.18. In the case of arbitrary deadlines, such a transition has to be explicitly added to the model, very much in the same way as the modelling of new task instantiations is done. In this case, the firing delay $Tick$ of $Clock$ is the greatest common divisor of the task periods *and* of their relative deadlines. Because the deadlines may be arbitrary, very often the value of $Tick$ will be 1. This leads to an increase in the number of tokens circulating in the CGPN model and implicitly to a potentially huge increase of the number of states of the underlying generalised semi-Markov process.

5.10 Conclusions

In the current and the previous chapter we have presented two approaches to the performance analysis of applications with stochastic task execution time. The first approach calculates the exact deadline miss ratios of tasks and task graphs and is efficient for monoprocessor systems. The second approach approximates the deadline miss ratios and is conceived for the complex case of multiprocessor systems.

While both approaches efficiently analyse one design alternative, they cannot be successfully applied to driving the optimisation phase of a system design process where a huge number of alternatives has to be evaluated. The next chapter presents a fast but less accurate analysis approach together with an approach for deadline miss ratio minimisation.

Chapter 6

Deadline Miss Ratio Minimisation

The previous two chapters addressed the problem of analysing the deadline miss ratios of applications with stochastic task execution times. In this chapter we address the complementary problem: given a multiprocessor hardware architecture and a functionality as a set of task graphs, find a task mapping and priority assignment such that the deadline miss ratios are below imposed upper bounds.

6.1 Problem Formulation

The problem addressed in this chapter is formulated as follows.

6.1.1 Input

The problem input consists of

- The set of processing elements PE, the set of buses B, and their connection to processors,
- The set of task graphs Γ,
- The set of task periods Π_T and the set of task graph periods Π_Γ,
- The set of task deadlines Δ_T and of task graph deadlines Δ_Γ,
- The set PE_τ of allowed mappings of τ for all tasks $\tau \in T$,
- The set of execution (communication) time probability density functions corresponding to each processing element $p \in PE_\tau$ for each task τ,
- The late task policy is the discarding policy,
- The set $Bounds = \{b_i \in \mathbb{N}\backslash\{0\} : 1 \leq i \leq g\}$, where $b_i = 1$, $\forall 1 \leq i \leq g$, i.e. there exists at most one active instantiation of any task graph in the system at any time,
- The set of task deadline miss thresholds Θ_T and the set of task graph deadline miss thresholds Θ_Γ, and
- The set of tasks and task graphs that are designated as being critical.

6.1.2 Output

The problem output consists of a mapping and priority assignment such that the cost function

$$\sum dev = \sum_{i=1}^{N} dev_{\tau_i} + \sum_{i=1}^{g} dev_{\Gamma_i} \tag{6.1}$$

giving the sum of miss deviations is minimised, where the deadline miss deviation is defined as in Section 3.2.5. If a mapping and priority assignment is found such that $\sum dev$ is finite, it is guaranteed that the deadline miss ratios of all critical tasks and task graphs are below their imposed thresholds.

6.1.3 Limitations

We restrict our assumptions on the system to the following:

- The scheduling policy is restricted to fixed priority non-preemptive scheduling.
- At most one instance of a task graph may be active in the system at any time.
- The late task policy is the discarding policy.

6.2 Approach Outline

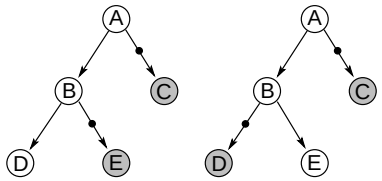
Because the defined problem is NP-hard (see the complexity of the classical mapping problem [GJ79]), we have to rely on heuristic techniques for solving the formulated problem. An accurate estimation of the miss deviation, which is used as a cost function for the optimisation process, is in itself a complex and time consuming task, as shown in the previous two chapters. Therefore, a fast approximation of the cost function value is needed to guide the design space exploration. Hence, the following subproblems have to be solved:

- Find an efficient design space exploration strategy, and
- Develop a fast and sufficiently accurate analysis, providing the needed cost indicators.

Section 6.4 discusses the first subproblems while Section 6.5 focuses on the system analysis we propose. First, however, we will present a motivation to our endeavour showing how naïve approaches fail to successfully solve the formulated problem.

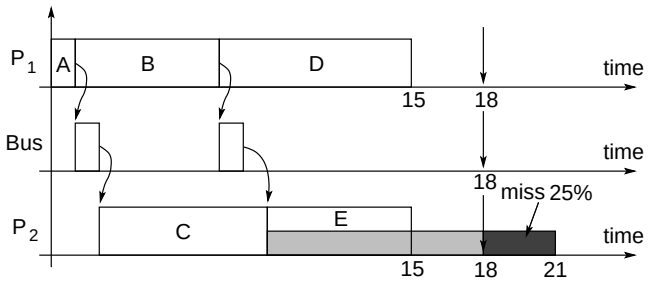
6.3 The Inappropriateness of Fixed Execution Time Models

A naïve approach to the formulated problem would be to use fixed execution time models (average, median, worst-case execution time, etc.) and to hope that the resulting designs would be optimal or close to optimal also from the point of view of the percentage of missed deadlines. The following example illustrates the pitfalls of such an approach and emphasises the need for an optimisation technique that considers

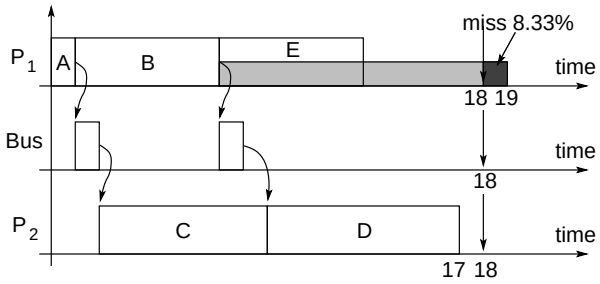


(a) (b)

Figure 6.1: Motivational example



(a)



(b)

Figure 6.2: Gantt diagrams of the two mapping alternatives in Figure 6.1

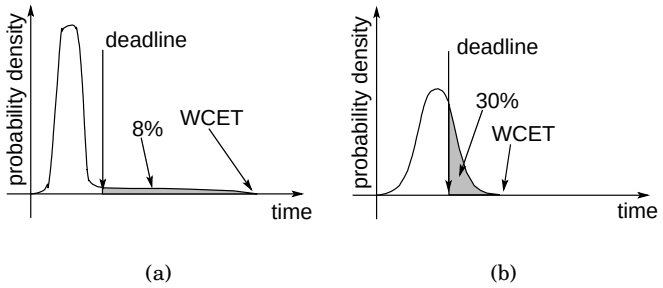


Figure 6.3: Motivational example

the stochastic execution times. Let us consider the application in Figure 6.1(a). All the tasks have period 20 and the deadline of the task graph is 18. Tasks A, B, C, and D have constant execution times of 1, 6, 7, and 8 respectively. Task E has a variable execution time whose probability is uniformly distributed between 0 and 12. Hence, the average (expected) execution time of task E is 6. The inter-processor communication takes 1 time unit per message. Let us consider the two mapping alternatives depicted in Figure 6.1(a) and 6.1(b). The two Gantt diagrams in Figure 6.2(a) and 6.2(b) depict the execution scenarios corresponding to the two considered mappings if the execution of task E took the expected amount of time, that is 6. The shaded rectangles depict the probabilistic execution of E. A mapping strategy based on the average execution times would select the mapping in Figure 6.1(a) as it leads to a shorter response time (15 compared to 17). However, in this case, the worst-case execution time of the task graph is 21. The deadline miss ratio of the task graph is $3/12 = 25\%$. If we took into consideration the stochastic nature of the execution time of task E, we would prefer the second mapping alternative, because of the better deadline miss ratio of $1/12 = 8.33\%$. If we considered worst-case response times instead of average ones, then we would choose the second mapping alternative, the same as the stochastic approach. However, approaches based on worst-case execution times can be dismissed by means of very simple counter-examples.

Let us consider a task τ that can be mapped on processor P_1 or on processor P_2. P_1 is a fast processor with a very deep pipeline. Because of its pipeline depth, mispredictions of target addresses of conditional jumps, though rare, are severely penalised. If τ is mapped on P_1, its ETPDF is shown in Figure 6.3(a). The long and flat density tail corresponds to the rare but expensive jump target address misprediction. If τ is mapped on processor P_2, its ETPDF is shown in Figure 6.3(b). Processor P_2 is slower with a shorter pipeline. The WCET of task τ on processor P_2 is smaller than the WCET if τ ran on processor P_1. Therefore, a design space exploration tool based on the WCET would map task τ on P_2. However, as Figure 6.3 shows, the deadline miss ratio in this case is larger than if task τ was mapped on processor P_1.

6.4 Mapping and Priority Assignment Heuristic

In this section, we propose a design space exploration strategy that maps tasks to processors and assigns priorities to tasks in order to minimise the cost function defined in Eq.(6.1). The exploration strategy is based on the Tabu Search (TS) heuristic [Glo89].

6.4.1 The Tabu Search Based Heuristic

Tabu Search is a heuristic introduced by Glover [Glo89]. We use an extended variant, which is described in this section. The variant is not specific to a particular problem. After explaining the heuristic in general, we will become more specific at the end of the section where we illustrate the heuristic in the context of task mapping and priority assignment.

Typically, optimisation problems are formulated as follows: Find a configuration, i.e. an assignment of values to parameters that characterise a system, such that the configuration satisfies a possibly empty set of imposed constraints and the value of a cost function is minimal for that configuration.

We define the design space $\mathcal{S}$ as a set of points (also called solutions), where each point represents a configuration that satisfies the imposed constraints. A *move* from one solution in the design space to another solution is equivalent to assigning a new value to one or more of the parameters that characterise the system. We say that we obtain solution s_2 by applying the move m on solution s_1, and we write $s_2 = m(s_1)$. Solution s_1 can be obtained back from solution s_2 by applying the *negated move* m, denoted $\overline{m}$ ($s_1 = \overline{m}(s_2)$).

Solution s' is a *neighbour* of solution s if there exists a move m such that solution s' can be obtained from solution s by applying move m. All neighbours of a solution s form the neighbourhood $V(s)$ of that solution ($V(s) = \{q : \exists m$ such that $q = m(s)\}$).

The exploration algorithm is shown in Figure 6.4. The exploration starts from an initial solution, labelled also as the current solution (line 1) considered as the globally best solution so far (line 2). The cost function is evaluated for the current solution (line 3). We keep track of a list of moves TM that are marked as *tabu*. Initially the list is empty (line 4).

We construct CM, a subset of the set of all moves that are possible from the current solution point (line 7). Let $N(CM)$ be the set of solutions that can be reached from the current solution by means of a move in CM.[1] The cost function is evaluated for each solution in $N(CM)$. A move $m \in CM$ is selected (line 8) if

- m is non-tabu and leads to the solution with the lowest cost among the solutions in $N(CM \backslash TM)$ ($m \notin TM \wedge cost(m(crt\_sol)) \leq cost(q), \forall q \in N(CM \backslash TM)$), or
- it is tabu but improves the globally best solution so far ($m \in TM \wedge cost(m(crt\_sol)) \leq global\_best$), or

[1] $N(CM) = V(crt\_sol)$ if CM is the set of *all* possible moves from crt_sol.

(1) $crt\_sol = init\_sol$
(2) $global\_best\_sol = crt\_sol$
(3) $global\_best = cost(crt\_sol)$
(4) $TM = \varnothing$
(5) $since\_last\_improvement = 0$
(6) $iteration\_count = 1$
(7) $CM = set\_of\_candidate\_moves(crt\_sol)$
(8) $(chose\_move, next\_sol\_cost) = choose\_move(CM)$
(9) **while** $iteration\_count < max\_iterations$ **do**
(10) **while** $since\_last\_improvement < W$ **do**
(11) $next\_sol = move(crt\_sol, chosen\_move)$
(12) $TM = TM \cup \{\overline{chosen\_move}\}$
(13) $since\_last\_improvement++$
(14) $iteration\_count++$
(15) $crt\_sol = next\_sol$
(16) **if** $next\_sol\_cost < global\_best\_cost$ **then**
(17) $global\_best\_cost = next\_sol\_cost$
(18) $global\_best\_sol = crt\_sol$
(19) $since\_last\_improvement = 0$
(20) **end if**
(21) $CM = set\_of\_candidate\_moves(TM, crt\_sol)$
(22) $(chosen\_move, next\_sol\_cost) = choose\_move(CM)$
(23) **end while**
(24) $since\_last\_improvement = 0$
(25) $(chosen\_move, next\_sol\_cost) = diversify(TM, crt\_sol)$
(26) $iteration\_count++$
(27) **end while**
(28) **return** $global\_best\_sol$

Figure 6.4: Design space exploration algorithm

- all moves in CM are tabu and m leads to the solution with the lowest cost among those solutions in $N(CM)$ ($\forall mv \in CM$, $mv \in TM \wedge cost(m(crt\_sol)) \leq cost(mv(crt\_sol))$).

The new solution is obtained by applying the chosen move m on the current solution (line 11). The reverse of move $\overline{m}$ is marked as tabu such that m will not be reversed in the next few iterations (line 12). The new solution becomes the current solution (line 15). If it is the case (line 16), the new solution becomes also the globally best solution reached so far (lines 17–18). However, it should be noted that the new solution could have a larger cost than the current solution. This could happen if there are no moves that would improve on the current solution or all such moves would be tabu. The list TM of tabu moves ensures that the heuristic does not get stuck in local minima. The procedure of building the set of candidate moves and then choosing one according to the criteria listed above is repeated. If no global improvement has been noted for the past W iterations, the loop (lines 10–23) is interrupted (line 10). In this case, a diversification phase follows (line 25) in which a rarely used move is performed in order to force the heuristic to explore different regions in the design space. The whole procedure is repeated until the heuristic iterated for a specified maximum number of iterations (line 9). The procedure returns the solution characterised by the lowest cost function value that it found during the design space exploration (line 28).

Two issues are of utmost importance when tailoring the general tabu search based heuristic described above for particular problems.

First, there is the definition of what is a legal move. On one hand, the transformation of a solution must result in another solution, i.e. the resulting parameter assignment must satisfy the set of constraints. On the other hand, because of complexity reasons, certain restrictions must be imposed on what constitutes a legal move. For example, if any transformation were a legal move, the neighbourhood of a solution would comprise the entire solution space. In this case, it is sufficient to run the heuristic for just one iteration ($max\_iterations = 1$) but that iteration would require an unreasonably long time, as the whole solution space would be probed. Nevertheless, if moves were too restricted, a solution could be reached from another solution only after applying a long sequence of moves. This makes the reaching of the far-away solution unlikely. In this case, the heuristic would be inefficient as it would circle in the same region of the solution space until a diversification step would force it out.

The second issue is the construction of the subset of candidate moves. One solution would be to include all possible moves from the current solution in the set of candidate moves. In this case, the cost function, which sometimes can be computationally expensive, has to be calculated for all neighbours. Thus, we would run the risk to render the exploration slow. If we had the possibility to quickly assess which are promising moves, we could include only those in the subset of candidate moves.

For our particular problem, namely the task mapping and priority assignment, each task is characterised by two attributes: its mapping and its priority. In this context, a move in the design space is equivalent to changing one or both attributes of one single task.

In the following section we discuss the issue of constructing the subset of candidate moves.

6.4.2 Candidate Move Selection

The cost function is evaluated $|CM|$ times at each iteration, where $|CM|$ is the cardinality of the set of candidate moves. Let us consider that task τ, mapped on processor P_j, is moved to processor P_i and there are q_i tasks on processor P_i. Task τ can take one of $q_i + 1$ priorities on processor P_i. If task τ is not moved to a different processor, but only its priority is changed on processor P_j, then there are $q_j - 1$ possible new priorities. If we consider all processors, there are $M - 2 + N$ possible moves for each task τ, as shown in the equation

$$q_j - 1 + \sum_{\substack{i=1 \\ i \neq j}}^{M} (q_i + 1) = M - 2 + \sum_{i=1}^{M} q_i = M - 2 + N, \tag{6.2}$$

where N is the number of tasks and M is the number of processors. Hence, if all possible moves are candidate moves,

$$N \cdot (M - 2 + N) \tag{6.3}$$

moves are possible at each iteration. Therefore, a key to the efficiency of the algorithm is the intelligent selection of the set CM of candidate moves. If CM contained only those moves that had a high chance to drive the search towards good solutions, then fewer points would be probed, leading to a speed up of the algorithm.

In our approach, the set CM of candidate moves is composed of all moves that operate on a subset of tasks. Tasks are assigned scores and the chosen subset of tasks is composed of the first K tasks with respect to their score. Thus, if we included all possible moves that modify the mapping and/or priority assignment of only the K highest ranked tasks, we would reduce the number of cost function evaluations N/K times.

We illustrate the way the scores are assigned to tasks based on the example in Figure 6.1(a). As a first step, we identify the critical paths and the non-critical paths of the application. In general, we consider a path to be an ordered sequence of tasks $(\tau_1, \tau_2, \ldots, \tau_n)$ such that τ_{i+1} is data dependent on τ_i. The average execution time of a path is given by the sum of the average execution times of the tasks belonging to the path. A path is critical if its average execution time is the largest among the paths belonging to the same task graph. For the example in Figure 6.1(a), the critical path is $A \rightarrow B \rightarrow D$, with an average execution time of $1 + 6 + 8 = 15$. In general, non-critical paths are those paths starting with a root node or a task on a critical path, ending with a leaf node or a task on a critical path and containing only tasks that do not belong to any critical path. For the example in Figure 6.1(a), non-critical paths are $A \rightarrow C$ and $B \rightarrow E$.

For each critical or non-critical path, a path mapping vector is computed. The mapping vector is a P-dimensional integer vector, where P is the number of processors. The modulus of its projection along dimension p_i is equal to the number of tasks that are mapped on processor p_i and that belong to the considered path. For the example in Figure 6.1(a), the vectors corresponding to the paths $A \rightarrow B \rightarrow D$, $A \rightarrow C$

and $B \rightarrow E$ are $3\mathbf{i} + 0\mathbf{j}$, $1\mathbf{i} + 1\mathbf{j}$, and $1\mathbf{i} + 1\mathbf{j}$ respectively, where **i** and **j** are the versors along the two dimensions. Each task is characterised by its task mapping vector, which has a modulus of 1 and is directed along the dimension corresponding to the processor on which the task is mapped. For example, the task mapping vectors of A, B, C, D, and E are $1\mathbf{i}$, $1\mathbf{i}$, $1\mathbf{j}$, $1\mathbf{i}$, and $1\mathbf{j}$ respectively.

Next, for each path and for each task belonging to that path, the angle between the path and the task mapping vectors is computed. For example, the task mapping vectors of tasks A, B, and D form an angle of 0° with the path mapping vector of critical path $A \rightarrow B \rightarrow D$ and the task mapping vectors of task A and C form an angle of 45° with the path mapping vector of the non-critical path $A \rightarrow C$. The score assigned to each task is a weighted sum of angles between the task's mapping vector and the mapping vectors of the paths to whom the task belongs. The weights are proportional to the relative criticality of the path. Intuitively, this approach attempts to map the tasks that belong to critical paths on the same processor. In order to avoid processor overload, the scores are penalised if the task is intended to be moved on highly loaded processors.

Once scores have been assigned to tasks, the first $K = N/c$ tasks are selected according to their scores. In our experiments, we use $c = 2$. In order to further reduce the search neighbourhood, not all possible moves that change the task mapping and/or priority assignment of one task are chosen. Only 2 processors are considered as target processors for each task. The selection of those two processors is made based on scores assigned to processors. These scores are a weighted sum of potential reduction of interprocessor communication and processor load. The processor load is weighted with a negative weight, in order to penalise overload. For example, if we moved task C from the shaded processor to the white processor, we would reduce the interprocessor communication with 100%. However, as the white processor has to cope with an average work load of 15 units (the average execution times of tasks A, B, and D), the 100% reduction would be penalised with an amount proportional to 15.

On average, there will be N/M tasks on each processor. Hence, if a task is moved to a different processor, it may take $N/M + 1$ possible priorities on its new processor. By considering only $N/2$ tasks and only 2 processors for each task, we restrict the neighbourhood to

$$N/2 \cdot 2 \cdot (1 + N/M) = N \cdot (1 + N/M) \tag{6.4}$$

candidate moves on average, i.e. approximately

$$N \cdot (M - 2 + N)/(N \cdot (1 + N/M)) \approx M \text{ times.} \tag{6.5}$$

We will denote this method as the restricted neighbourhood search. In Section 6.6 we will compare the restricted neighbourhood search with an exploration of the complete neighbourhood.

6.5 Analysis

This section presents our approximate analysis algorithm. The first part of the section discusses the deduction of the algorithm itself, while the second part presents some considerations on the approximations that were made and their impact regarding the accuracy.

6.5.1 Analysis Algorithm

The cost function that is driving the design space exploration is $\sum dev$, where dev is the miss deviation as defined in Eq.(6.1). The miss deviation for each task is obtained as the result of a performance analysis of the system.

In the previous chapter, we presented a performance analysis method for multiprocessor applications with stochastic task executions times. The method is based on the Markovian analysis of the underlying stochastic process. As the latter captures all possible behaviours of the system, the method gives great insight regarding the system's internals and bottlenecks. However, its relatively large analysis time makes its use inappropriate inside an optimisation loop. Therefore, we propose an approximate analysis method of polynomial complexity. The main challenge is in finding those dependencies among random variables that are weak and can be neglected such that the analysis becomes of polynomial complexity and the introduced inaccuracy is within reasonable bounds.

Before proceeding with the exposition of the approximate analysis approach, we introduce the notation that we use in the sequel.

The finishing time of the j^{th} job of task τ is the time moment when (τ, j) finishes its execution. We denote it with $F_{\tau,j}$. The deadline miss ratio of a job is the probability that its finishing time exceeds its deadline:

$$m_{\tau,j} = 1 - \mathsf{P}(F_{\tau,j} \leq \delta_{\tau,j}) \tag{6.6}$$

The ready time of (τ, j) is the time moment when (τ, j) is ready to execute, i.e. the maximum of the finishing times of jobs in its predecessor set. We denote the ready time with $A_{\tau,j}$ and we write

$$A_{\tau,j} = \max_{\sigma \in {}^\circ\tau} F_{\sigma,j} \tag{6.7}$$

The starting time of (τ, j) is the time moment when (τ, j) starts executing. We denote it with $S_{\tau,j}$. Obviously, the relation

$$F_{\tau,j} = S_{\tau,j} + Ex_{\tau,j} \tag{6.8}$$

holds between the starting and finishing times of (τ, j), where $Ex_{\tau,j}$ denotes the execution time of (τ, j). The ready time and starting times of a job may differ because the processor might be busy at the time the job becomes ready for execution. The ready, starting and finishing times are all random variables.

Let $L_{\tau,j}(t)$ be a function that takes value 1 if (τ, j) is running at time moment t and 0 otherwise. In other words, if $L_{\tau,j}(t) = 1$, processing element $Map(\tau)$ is busy executing job j of task τ at time t. If (τ, j) starts executing at time t, $L_{\tau,j}(t)$ is considered to be 1. If (τ, j) finishes its execution at time t', $L_{\tau,j}(t')$ is considered to be 0. For simplicity, in the sequel, we will write $L_{\tau,j}(t)$ when we mean $L_{\tau,j}(t) = 1$. Also, $L_\sigma(t)$ is a shorthand notation for $\sum_{j \in \mathbb{N}} L_{\sigma,j}(t)$.

Let $I_{\tau,j}(t)$ be a function that takes value 1 if

- All tasks in the ready-to-run queue of the scheduler on processor $Map(\tau)$ at time t have a lower priority than task τ, and
- $\sum_{\sigma \in \mathcal{T}_\tau \setminus \{\tau\}} L_\sigma(t) = 0$,

and it takes value 0 otherwise, where $\mathcal{T}_\tau = \mathcal{T}_{Map(\tau)}$ is the set of tasks mapped on the same processor as task τ. Intuitively, $I_{\tau,j}(t) = 1$ implies that (τ, j) could start running on processing element $Map(\tau)$ at time t if (τ, j) becomes ready at or prior to time t. Let $I_{\tau,j}(t, t')$ be a shorthand notation for $\exists \xi \in (t, t'] : I_{\tau,j}(\xi) = 1$, i.e. there exists a time moment ξ in the right semi-closed interval $(t, t']$, such that (τ, j) could start executing at ξ if it become ready at or prior to ξ.

In order to compute the deadline miss ratio of (τ, j) (Eq.(6.6)), we need to compute the probability distribution of the finishing time $F_{\tau,j}$. This in turn can be precisely determined (Eq.(6.8)) from the probability distribution of the execution time $Ex_{\tau,j}$, which is an input data, and the probability distribution of the starting time of (τ, j), $S_{\tau,j}$. Therefore, in the sequel, we focus on determining $\mathsf{P}(S_{\tau,j} \leq t)$.

We start by observing that $I_{\tau,j}(t, t + h)$ is a necessary condition for $t < S_{\tau,j} \leq t + h$. Thus,

$$\mathsf{P}(t < S_{\tau,j} \leq t + h) = \mathsf{P}(t < S_{\tau,j} \leq t + h \cap I_{\tau,j}(t, t + h)). \tag{6.9}$$

We can write

$$\begin{aligned} \mathsf{P}(t < S_{\tau,j} \leq t + h \cap I_{\tau,j}(t, t + h)) = \\ = \mathsf{P}(t < S_{\tau,j} \cap I_{\tau,j}(t, t + h)) - \\ - \mathsf{P}(t + h < S_{\tau,j} \cap I_{\tau,j}(t, t + h)). \end{aligned} \tag{6.10}$$

Furthermore, we observe that the event

$$t + h < S_{\tau,j} \cap I_{\tau,j}(t, t + h)$$

is equivalent to

$$\begin{gathered} (t + h < A_{\tau,j} \cap I_{\tau,j}(t, t + h)) \cup \\ \cup (sup\{\xi \in (t, t + h] : I_{\tau,j}(\xi) = 1\} < A_{\tau,j} \leq t + h \cap I_{\tau,j}(t, t + h)). \end{gathered}$$

In other words, (τ, j) starts executing after $t+h$ when the processor was available sometimes in the interval $(t, t + h]$ if and only if (τ, j) became ready to execute *after* the latest time in $(t, t + h]$ at which the processor was available. Thus, we can rewrite Eq.(6.10) as follows:

$$\begin{aligned} &\mathsf{P}(t < S_{\tau,j} \leq t + h \cap I_{\tau,j}(t, t + h)) = \\ &= \mathsf{P}(t < S_{\tau,j} \cap I_{\tau,j}(t, t + h)) - \\ &- \mathsf{P}(t + h < A_{\tau,j} \cap I_{\tau,j}(t, t + h)) - \\ &- \mathsf{P}(sup\{\xi \in (t, t + h] : I_{\tau,j}(\xi) = 1\} < A \leq t + h \cap \\ &\quad \cap I_{\tau,j}(t, t + h)). \end{aligned} \tag{6.11}$$

After some manipulations involving negations of the events in the above equation, and by using Eq.(6.9), we obtain

$$\begin{aligned} &\mathsf{P}(t < S_{\tau,j} \leq t + h) = \mathsf{P}(A_{\tau,j} \leq t + h \cap I_{\tau,j}(t, t + h)) - \\ &- \mathsf{P}(S_{\tau,j} \leq t \cap I_{\tau,j}(t, t + h)) - \\ &- \mathsf{P}(sup\{\xi \in (t, t + h] : I_{\tau,j}(\xi) = 1\} < A_{\tau,j} \leq t + h \cap \\ &\quad \cap I_{\tau,j}(t, t + h)). \end{aligned} \tag{6.12}$$

When h becomes very small, the last term of the right-hand side of the above equation becomes negligible relative to the other two terms.

Hence, we write the final expression of the distribution of $S_{\tau,j}$ as follows:

$$\begin{aligned} &\mathsf{P}(t < S_{\tau,j} \leq t+h) \approx \\ &\approx \mathsf{P}(A_{\tau,j} \leq t+h \cap I_{\tau,j}(t,t+h)) - \\ &- \mathsf{P}(S_{\tau,j} \leq t \cap I_{\tau,j}(t,t+h)). \end{aligned} \tag{6.13}$$

We observe from Eq.(6.13) that the part between t and $t+h$ of the probability distribution of $S_{\tau,j}$ can be calculated from the probability distribution of $S_{\tau,j}$ for time values less than t. Thus, we have a method for an iterative calculation of $\mathsf{P}(S_{\tau,j} \leq t)$, in which we compute

$$\mathsf{P}(kh < S_{\tau,j} \leq (k+1)h), k \in \mathbb{N},$$

at iteration $k+1$ from values obtained during previous iterations.

A difficulty arises in the computation of the two joint distributions in the right-hand side of Eq.(6.13). The event that a job starts or becomes ready prior to time t *and* that the processor may start executing it in a vicinity of time t is a very complex event. It depends on many aspects, such as the particular order of execution of tasks on different (often all) processors, and on the execution time of different tasks, quite far from task τ in terms of distance in the computation tree. Particularly the dependence on the execution order of tasks on different processors makes the exact computation of

$$\mathsf{P}(A_{\tau,j} \leq t+h \cap I_{\tau,j}(t,t+h))$$

and

$$\mathsf{P}(S_{\tau,j} \leq t \cap I_{\tau,j}(t,t+h))$$

of exponential complexity. Nevertheless, exactly this multitude of dependencies of events $I_{\tau,j}(t,t+h)$, $A_{\tau,j} \leq t+h$, and $S_{\tau,j} \leq t$ on various events makes the dependency weak among the aforementioned three events. Thus, we approximate the right-hand side of Eq.(6.13) by considering the joint events as if they were conjunctions of independent events. Hence, we approximate $\mathsf{P}(t < S_{\tau,j} \leq t+h)$ as follows:

$$\begin{aligned} &\mathsf{P}(t < S_{\tau,j} \leq t+h) \approx \\ &\approx (\mathsf{P}(A_{\tau,j} \leq t+h) - \mathsf{P}(S_{\tau,j} \leq t)) \cdot \mathsf{P}(I_{\tau,j}(t,t+h)). \end{aligned} \tag{6.14}$$

The impact of the introduced approximation on the accuracy of the analysis is discussed in Section 6.5.2 based on a non-trivial example.

In order to fully determine the probability distribution of $S_{\tau,j}$ (and implicitly of $F_{\tau,j}$ and the deadline miss ratio), we need the probability distribution of $A_{\tau,j}$ and the probability $\mathsf{P}(I_{\tau,j}(t,t+h))$. Based on Eq.(6.7), if the finishing times of all tasks in the predecessor set of task τ were statistically independent, we could write

$$\mathsf{P}(A_\tau \leq t) = \prod_{\sigma \in {}^\circ\tau} \mathsf{P}(F_\sigma \leq t). \tag{6.15}$$

In the majority of cases the finishing times of all tasks in the predecessor set of task τ are not statistically independent. For example, if there exists a task α and two computation paths $\alpha \rightarrow \sigma_1$ and $\alpha \rightarrow \sigma_2$, where tasks σ_1 and σ_2 are predecessors of task τ, then the finishing times F_{σ_1} and F_{σ_2} are not statistically independent. The dependency

(1) Sort all tasks in topological order of the task graph and put the sorted tasks in sequence T
(2) For all (τ, j), such that τ has no predecessors determine $\mathsf{P}(A_{\tau,j} \leq t)$
(3) For all (τ, j), let $\mathsf{P}(I_{\tau,j}(0, h)) = 1$
(4) **for** $t := 0$ **to** LCM **step** h **do**
(5) **for each** $\tau \in T$ **do**
(6) compute $\mathsf{P}(A_\tau \leq t)$ Eq.(6.15)
(7) compute $\mathsf{P}(t < S_{\tau,j} \leq t + h)$ Eq.(6.14)
(8) compute $\mathsf{P}(F_{\tau,j} \leq t + h)$ Eq.(6.8)
(9) compute $\mathsf{P}(L_{\tau,j}(t + h))$ Eq.(6.17)
(10) compute $\mathsf{P}(I_{\tau,j}(t, t + h))$ Eq.(6.16)
(11) **end for**
(12) **end for**
(13) compute the deadline miss ratios Eq.(6.6)

Figure 6.5: Approximate analysis algorithm

becomes weaker the longer these computation paths are. Also, the dependency is weakened by the other factors that influence the finishing times of tasks σ, for example the execution times and execution order of the tasks on processors $Map(\sigma_1)$ and $Map(\sigma_2)$. Even if no common ancestor task exists among any of the predecessor tasks σ, the finishing times of tasks σ may be dependent because they or some of their predecessors are mapped on the same processor. However, these kind of dependencies are extremely weak as shown by Kleinrock [Kle64] for computer networks and by Li [LA97] for multiprocessor applications. Therefore, in practice, Eq.(6.15) is a good approximation.

Last, we determine the probability $\mathsf{P}(I_{\tau,j}(t, t + h))$, i.e. the probability that processor $Map(\tau)$ may start executing (τ, j) sometimes in the interval $(t, t + h]$. This probability is given by the probability that no task is executing at time t, i.e.

$$\mathsf{P}(I_{\tau,j}(t, t + h)) = 1 - \sum_{\sigma \in \mathcal{T}_\tau \setminus \{\tau\}} \mathsf{P}(L_\sigma(t) = 1). \tag{6.16}$$

The probability that (τ, j) is running at time t is given by

$$\mathsf{P}(L_{\tau,j}(t) = \mathsf{P}(S_{\tau,j} \leq t) - \mathsf{P}(F_{\tau,j} \leq t). \tag{6.17}$$

The analysis algorithm is shown in Figure 6.5. The analysis is performed over the interval $[0, LCM)$, where LCM is the least common multiple of the task periods. The algorithm computes the probability distributions of the random variables of interest parsing the set of tasks in their topological order. Thus, we make sure that ready times propagate correctly from predecessor tasks to successors.

Line 7 of the algorithm computes the probability that job (τ_i, j) starts its execution sometime in the interval $(t, t + h]$ according to Eq. (6.14). The finishing time of the job may lie within one of the intervals $(t + BCET_i, t + h + BCET_i]$, $(t + BCET_i + h, t + 2h + BCET_i]$, $\ldots$, $(t + WCET_i, t + h + WCET_i]$, where $BCET_i$ and $WCET_i$ are the best and worst-case execution times of task τ_i respectively. There are $\lceil (WCET_i - BCET_i)/h \rceil$ such intervals. Thus, the computation of the probability distribution of the finishing time of the task (line 8) takes $\lceil |ETPDF_i|/h \rceil$ steps, where $|ETPDF_i| = WCET_i - BCET_i$.

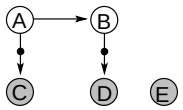


Figure 6.6: Application example

Let $|ETPDF| = \max_{1 \leq i \leq N} \lceil |ETPDF_i|/h \rceil$. Then the complexity of the algorithm is $O(N \cdot LCM/h \cdot \lceil |ETPDF|/h \rceil)$, where N is the number of processing and communication tasks. The choice of the discretisation resolution h is done empirically such that we obtain a fast analysis with reasonable accuracy for the purpose of task mapping.

6.5.2 Approximations

We have made several approximations in the algorithm described in the previous section. These are:

1. The discretisation approximation used throughout the approach, i.e. the fact that the probability distributions of interest are all computed at discrete times $\{0, h, 2h, \ldots, \lfloor LCM/h \rfloor\}$,
2. $\mathsf{P}(A_\tau \leq t) \approx \prod_{\sigma \in {}^\circ\tau} \mathsf{P}(F_\sigma \leq t)$,
3. $\mathsf{P}(A_{\tau,j} \leq t + h \cap I_{\tau,j}(t, t+h)) \approx \mathsf{P}(A_{\tau,j} \leq t + h) \cdot \mathsf{P}(I_{\tau,j}(t, t+h))$ and $\mathsf{P}(S_{\tau,j} \leq t \cap I_{\tau,j}(t, t+h)) \approx \mathsf{P}(S_{\tau,j} \leq t) \cdot \mathsf{P}(I_{\tau,j}(t, t+h))$.

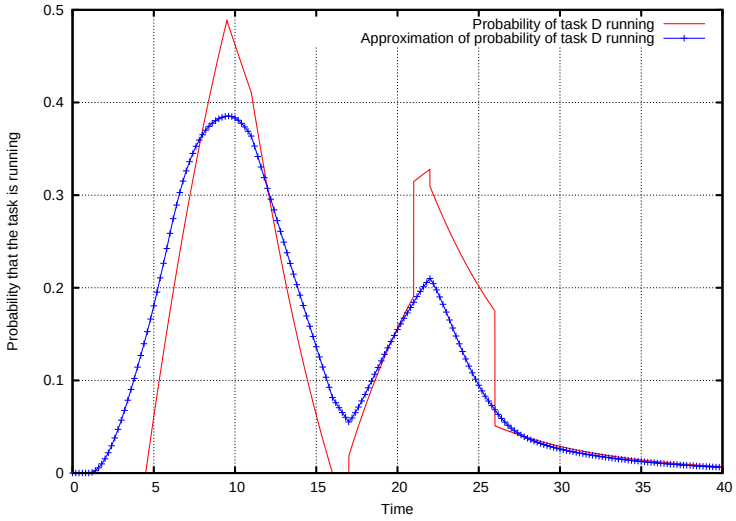
The first approximation is inevitable when dealing with continuous functions. Moreover, its accuracy may be controlled by choosing different discretisation resolutions h.

The second approximation is typically accurate as the dependencies between the finishing times F_σ are very weak [Kle64], and we will not focus on its effects in this discussion.

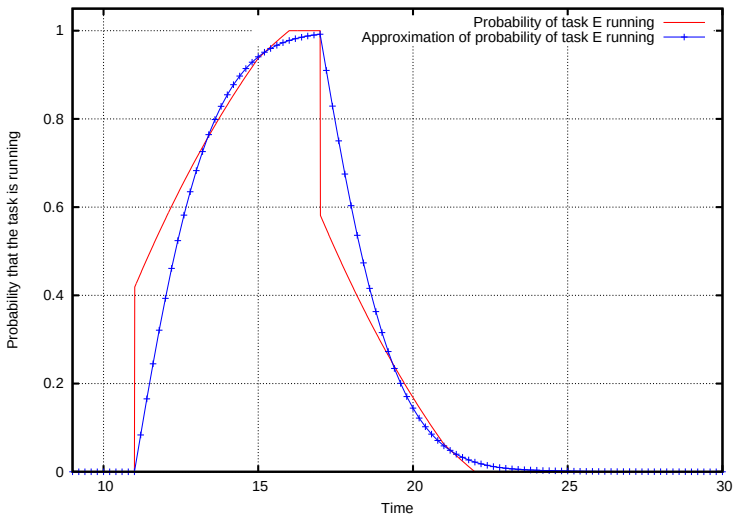
In order to discuss the last approximation, we will introduce the following example. Let us consider the application depicted in Figure 6.6. It consists of 5 tasks, grouped into two task graphs $\Gamma_1 = \{A, B, C, D\}$ and $\Gamma_2 = \{E\}$. Tasks A and B are mapped on the first processor, while tasks C, D, and E are mapped on the second processor. The two black dots on the arrows between tasks A and C, and tasks B and D represent the inter-processor communication tasks. Tasks C, D, and E have fixed execution times of 4, 5, and 6 time units respectively. Tasks A and B have execution times with exponential probability distributions, with average rates of $1/7$ and $1/2$ respectively.[2] Each of the two inter-processor communications takes 0.5 time units. Task A arrives at time moment 0, while task E arrives at time moment 11. Task E is the highest priority task. The deadline of both task graphs is 35.

Because of the data dependencies between the tasks, task D is the last to run among the task graph Γ_1. The probability that processor two is executing task D at time t is analytically determined and plotted in Figure 6.7(a) as a function of t. On the same figure, we plotted the approximation of the same probability as obtained by our approximate

[2]We chose exponential execution time probability distributions only for the scope of this illustrative example. Thus, we are able to easily deduce the exact distributions in order to compare them to the approximated ones. Note that our approach is not restricted to exponential distributions and we use generalised distributions throughout the experimental results



(a) Approximation of the probability that task D is running



(b) Approximation of the probability that task E is running

Figure 6.7: Approximation accuracy

Task	Average error	Standard deviation of errors
19	0.056351194	0.040168796
13	0.001688039	0.102346107
5	0.029250265	0.178292338
9	0.016695770	0.008793487

Table 6.1: Approximation accuracy

analysis method. The probability that task E is running at time t and its approximation are shown in Figure 6.7(b).

In Figure 6.7(a), we observe that large approximation errors occur at times around the earliest possible start time of task D, i.e. around time 4.5.[3] We can write

$$\begin{aligned}&\mathsf{P}(A_D \leq t+h \cap I_D(t,t+h)) = \\ &= \mathsf{P}(I_D(t,t+h)|A_D \leq t+h) \cdot \mathsf{P}(A_D \leq t+h).\end{aligned}$$

$\mathsf{P}(I_D(t,t+h)|A_D \leq t+h)$ is interpreted as the probability that task D may start to run in the interval $(t,t+h]$ *knowing* that it became ready to execute prior to time $t+h$. If $t+h < 4.5$, and we took into consideration the fact that $A_D \leq t+h$, then we know for sure that task C could not have yet finished its execution of 4 time units (see footnote). Therefore

$$\mathsf{P}(I_D(t,t+h)|A_D \leq t+h) = 0, t+h < 4.5.$$

However, in our analysis, we approximate $\mathsf{P}(I_D(t,t+h)|A_D \leq t+h)$ with $\mathsf{P}(I_D(t,t+h))$, i.e. we do *not* take into account that $A_D \leq t$. Not taking into account that task D became ready prior to time t opens the possibility that task A has not yet finished its execution at time t. In this case, task C has not yet become ready, and the processor on which tasks C and D are mapped could be idle. Thus,

$$\mathsf{P}(I_D(t,t+h)) \neq 0,$$

because the processor might be free if task C has not yet started. This illustrates the kind of approximation errors introduced by

$$\mathsf{P}(A_D \leq t+h \cap I_D(t,t+h)) \approx \mathsf{P}(I_D(t,t+h)) \cdot \mathsf{P}(A_D \leq t+h).$$

However, what we are interested in is a high-quality approximation towards the *tail* of the distribution, because typically there is the deadline. As we can see from the plots, the two curves almost overlap for $t > 27$. Thus, the approximation of the deadline miss ratio of task D is very good. The same conclusion is drawn from Figure 6.7(b). In this case too we see a perfect match between the curves for time values close to the deadline.

Finally, we assessed the quality of our approximate analysis on larger examples. We compare the processor load curves obtained by our approximate analysis (AA) with processor load curves obtained by our high-complexity performance analysis (PA) presented in the previous chapter. The benchmark application consists of 20 processing tasks

[3]The time when task D becomes ready is always *after* the time when task C becomes ready. Task C is ready the earliest at time 0.5, because the communication $A \rightarrow C$ takes 0.5 time units. The execution of task C takes 4 time units. Therefore, the processor is available to task D the earliest at time 4.5.

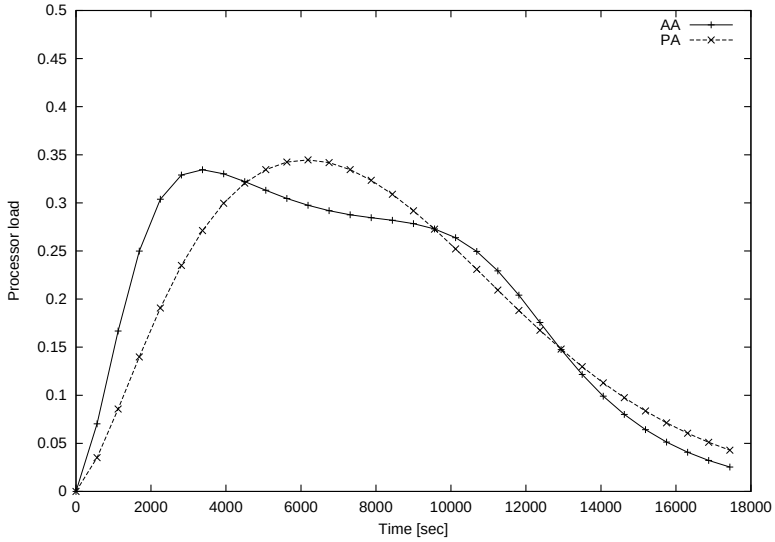


Figure 6.8: Approximation accuracy

mapped on 2 processors and 3 communication tasks mapped on a bus connecting the two processors. Figure 6.8 gives a qualitative measure of the approximation. It depicts the two processor load curves for a task in the benchmark application. One of the curves was obtained with PA and the other with AA. A quantitative measure of the approximation is given in Table 6.1. We present only the extreme values for the average errors and standard deviations. Thus, row 1 in the table, corresponding to task 19, shows the largest obtained average error, while row 2, corresponding to task 13, shows the smallest obtained average error. Row 3, corresponding to task 5, shows the worst obtained standard deviation, while row 4, corresponding to task 9, shows the smallest obtained standard deviation. The average of standard deviations of errors over all tasks is around 0.065. Thus, we can say with 95% confidence that AA approximates the processor load curves with an error of ± 0.13.

6.6 Experimental Results

The proposed heuristic for task mapping and priority assignment has been experimentally evaluated on randomly generated benchmarks and on a real-life example. This section presents the experimental setup and comments on the obtained results. The experiments were run on a desktop PC with an AMD Athlon processor clocked at 1533MHz.

The benchmark set consisted of 396 applications. The applications contained t tasks, clustered in g task graphs and mapped on p processors, where $t \in \{20, 22, \ldots, 40\}$, $g \in \{3, 4, 5\}$, and $p \in \{3, 4, \ldots, 8\}$. For each combination of t, g, and p, two applications were randomly generated. Three mapping and priority assignment methods were run on each application. All three implement a Tabu Search algorithm with the same tabu tenure, termination criterion and number of iterations after which a diversification phase occurs. In each iteration, the first method selects the next point in the design space while considering the

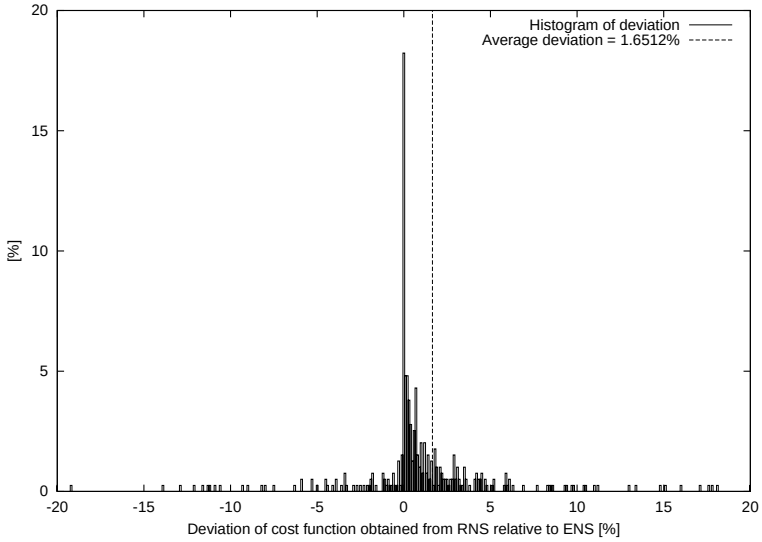


Figure 6.9: Cost obtained by RNS vs. ENS

entire neighbourhood of design space points. Therefore, we denote it ENS, exhaustive neighbourhood search. The second method considers only a restricted neighbourhood of design space points when selecting the next design transformation. The restricted neighbourhood is defined as explained in Section 6.4. We call the second method RNS, restricted neighbourhood search. Both ENS and RNS use the same cost function, defined in Eq.(6.1) and calculated according to the approximate analysis described in Section 6.5. The third method considers only fixed task execution times, equal to the average task execution times. It uses an exhaustive neighbourhood search and minimises the value of the cost function $\sum lax_\tau$, where lax_τ is defined as follows

$$lax_\tau = \begin{cases} \infty & F_\tau > \delta_\tau \wedge \tau \text{ is critical} \\ F_\tau - \delta_\tau & \text{otherwise.} \end{cases} \tag{6.18}$$

The third method is abbreviated LO-AET, laxity optimisation based on average execution times. Once LO-AET has produced a solution, the cost function defined in Eq.(6.1) is calculated and reported for the produced mapping and priority assignment.

6.6.1 RNS and ENS: Quality of Results

The first issue we look at is the quality of results obtained with RNS compared to those produced by ENS. The deviation of the cost function obtained from RNS relative to the cost function obtained by ENS is defined as

$$\frac{cost_{RNS} - cost_{ENS}}{cost_{ENS}} \tag{6.19}$$

Figure 6.9 depicts the histogram of the deviation over the 396 benchmark applications. The relative deviation of the cost function appears on the x-axis. The value on the y-axis corresponding to a value x on the x-axis indicates the percentage of the 396 benchmarks that have a cost function deviation equal to x. On average, RNS is only 1.65% worse

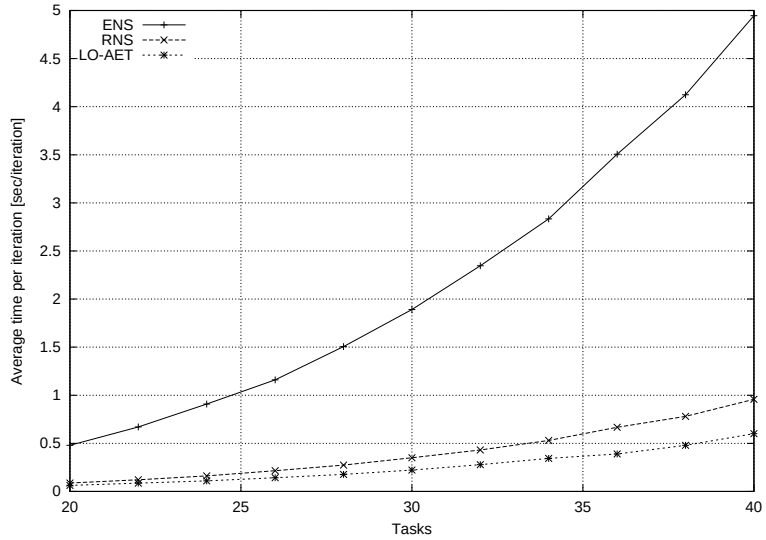


Figure 6.10: Run times of RNS vs. ENS

than ENS. In 19% of the cases, the obtained deviation was between 0 and 0.1%. Note that RNS can obtain better results than ENS (negative deviation). This is due to the intrinsically heuristic nature of Tabu Search.

6.6.2 RNS and ENS: Exploration Time

As a second issue, we compared the run times of RNS, ENS, and LO-AET. Figure 6.10 shows the average times needed to perform one iteration in RNS, ENS, and LO-AET respectively. It can be seen that RNS runs on average 5.16–5.6 times faster than ENS. This corresponds to the theoretical prediction, made at in Section 6.4.2, stating that the neighbourhood size of RNS is M times smaller than the one of ENS when $c = 2$. In our benchmark suite, M is between 3 and 8 averaging to 5.5. We also observe that the analysis time is close to quadratic in the number of tasks, which again corresponds to the theoretical result that the size of the search neighbourhood is quadratic in N, the number of tasks.

We finish the Tabu Search when $40 \cdot N$ iterations have executed, where N is the number of tasks. In order to obtain the execution times of the three algorithms, one needs to multiply the numbers on the ordinate in Figure 6.10 with $40 \cdot N$. For example, for 40 tasks, RNS takes circa 26 minutes while ENS takes roughly 2h12'.

6.6.3 RNS and LO-AET: Quality of Results and Exploration Time

The LO-AET method is marginally faster than RNS. However, as shown in Figure 6.11, the value of the cost function obtained by LO-AET is on average almost an order of magnitude worse (9.09 times) than the one obtained by RNS. This supports one of the main messages of this chapter, namely that considering a fixed execution time model for optimisation of systems is completely unsuitable if deadline miss

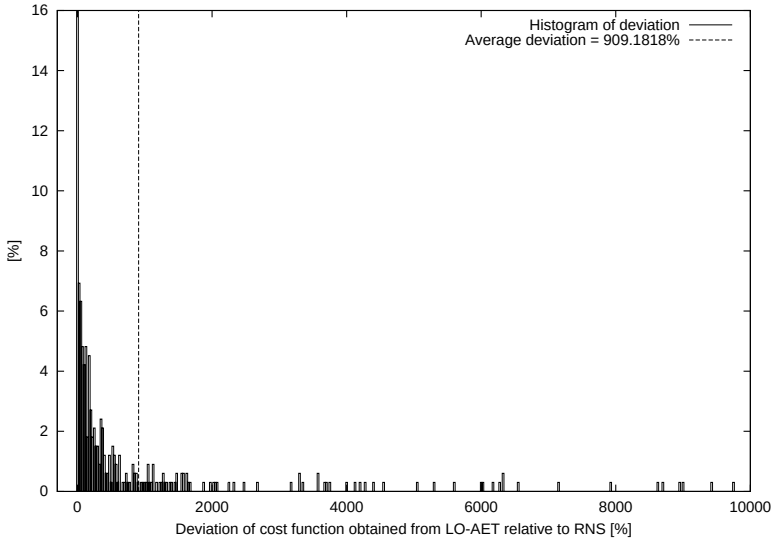


Figure 6.11: Cost obtained by LO-AET vs. RNS

ratios are to be improved. Although LO-AET is able to find a good implementation in terms of average execution times, it turns out that this implementation is very poor from the point of view of deadline miss ratios. What is needed is a heuristic like RNS, which is explicitly driven by deadline miss ratios during design space exploration.

6.6.4 Real-Life Example: GSM Voice Decoding

Last, we considered an industrial-scale real-life example from the telecommunication area, namely a smart GSM cellular phone [Sch03], containing voice encoder and decoder, an MP3 decoder, as well as a JPEG encoder and decoder.

In GSM a second of human voice is sampled at 8kHz, and each sample is encoded on 13 bits. The resulting stream of 13000 bytes per second is then encoded using so-called regular pulse excitation long-term predictive transcoder (GSM 06.10 specification [ETS]). The encoded stream has a rate of 13000 *bits* per second, i.e. a frame of 260 bits arrives every 20ms. Such a frame is decoded by the application shown in Figure 6.12. It consists of one task graph of 34 tasks mapped on two processors. The task partitioning and profiling was done by M. Schmitz [Sch03]. The period of every task is equal to the frame period, namely 20ms. The tasks process an input block of 260 bits. The layout of a 260 bit frame is shown on the top of Figure 6.12, where also the correspondence between the various fields in the frame and the tasks processing them is depicted.

For all tasks, the deadline is equal to the period. No tasks are critical in this application but the deadline miss threshold of every task is 0. Hence, the value of the cost function defined in Eq.(6.1) is equal to the sum of the deadline miss ratios of all 34 tasks and the deadline miss ratio of the entire application.

The restricted neighbourhood search found a task mapping and priority assignment of cost 0.0255 after probing $729,662$ potential solutions in $1\text{h}31'$ on an AMD Athlon clocked at 1533MHz. This means that the

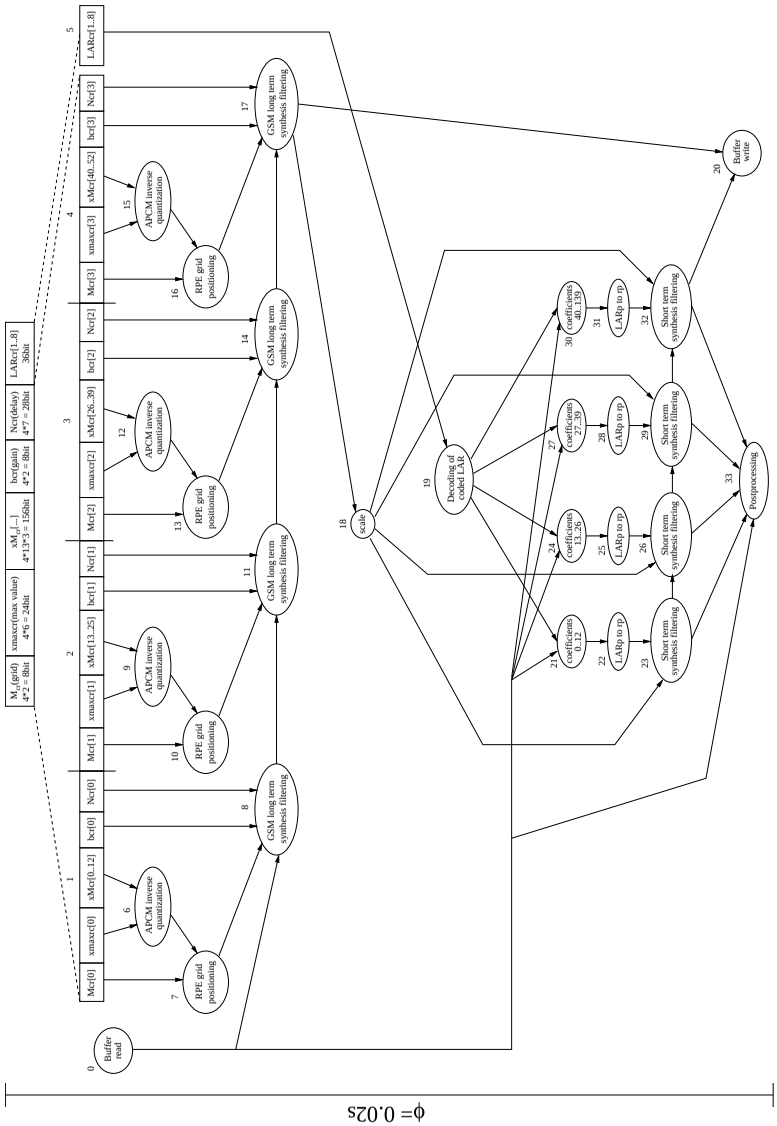


Figure 6.12: Task graph modelling GSM voice decoding. From M. Schmitz's [Sch03] PhD thesis.

deadline miss ratio of the voice decoding application, if the tasks are mapped and their priority is assigned as found by the RNS, is less than 2.55%. This result is about 16 times better than the cost of an initial random solution.

Part III

Communication Synthesis for Networks-on-Chip

Chapter 7

Motivation and Related Work

Transient failures of on-chip network links are a source of stochastic behaviour of applications implemented on networks-on-chip. In this chapter, we introduce the motivation of our work in the area of providing reliable and low-energy communication under timeliness constraints. Next, in Section 7.2 we survey the related work in the area and underline our contributions.

7.1 Motivation

Shrinking feature sizes make possible the integration of millions and soon billions of transistors on multi-core chips. At this integration level, effects such as capacitive cross-talk, power supply noise, and neutron and alpha radiation [SN96, AKK+00] lead to non-negligible rates of transient failures of interconnects and/or devices, jeopardising the correctness of applications.

For example, new technologies such as Extreme Ultraviolet Lithography promise to deliver feature sizes of 20nm [Die00]. This allows for single-chip implementations of extremely complex, computation-intensive applications, such as advanced signal processing in, for example, the military or medical domains, high-quality multimedia processing, high-throughput network routing, and high-traffic web services.

However, these technological capabilities do not come without unprecedented challenges to the design community. These challenges include increased design and verification complexity and high power density and an increased rate of transient faults of the components and/or communication links.

Several authors [BD02, DRGR03, KJS+02] have proposed network-on-chip (NoC) architectures as replacements to bus-based designs in order to improve scalability, reduce design, verification and test complexity and to ease the power management problem.

With shrinking feature size, the on-chip interconnects have become a performance bottleneck [Dal99]. Thus, a first concern, which we address in this part of the thesis, is *application latency*.

The energy consumption of wires has been reported to account for about 40% of the total energy consumed by the chip [Liu94]. Moreover, another significant source of energy consumption are the buffers

distributed within the on-chip communication infrastructure. This is a strong incentive to consider the communication energy reduction by means of efficient utilisation of the on-chip communication channels and by means of reducing the buffering demand of applications. Thus, a second concern, which we address in Chapters 9 and 10, is communication energy and buffer space demand minimisation.

A third problem arising from shrinking feature size is the increasing rate of transient failures of the communication lines. The reliability of network nodes is guaranteed by specific methods, which are outside the scope of this work. In general, 100% reliable communication cannot be achieved in the presence of transient failures, except under assumptions such as no multiple simultaneous faults or at most n bit flips, which are unrealistic in the context of complex NoC. Hence, we are forced to tolerate occasional errors, provided that they occur with a rate below an imposed threshold. Thus, a third concern, addressed in Chapter 9, is to ensure an imposed communication reliability degree under constraints on application latency, while keeping energy consumption as low as possible.

We address the three identified problems, namely energy reduction and satisfaction of timeliness and communication reliability constraints, by means of *communication synthesis*. In this context, synthesizing the communication means mapping data packets to network links and determining the time moments when the packets are released on the links. The selection of message routes has a significant impact on the responsiveness of applications implemented on the NoC. The communication reliability is ensured by deploying a combination of spatially and temporally redundant communication. This however renders the communication mapping problem particularly difficult.

The next section surveys related work and contrasts it with ours. Chapter 9 presents our approach to communication mapping for energy-efficient reliable communication with predictable latency. Chapter 10 presents a communication synthesis approach for the minimisation of the buffer space demands of applications.

7.2 Related Work

Communication synthesis greatly affects performance and energy consumption. Closest to our approach, which maps data packets to network links in an off-line manner, is deterministic routing [HM05, MD04]. One of its advantages is that it may guarantee deadlock-free communication and the communication latency and energy consumption are easier to predict. Nevertheless, deterministic routing can be efficiently applied only if traffic patterns are known in more detail at design time. Under the assumptions that we make in this thesis, the communication mapping (and the deterministic routing that results from it) is complicated by the fact that we deploy redundant communication.

Wormhole routing [BIGA04] is a popular switching technique among NoC designs. However, an analysis that would provide bounds on its latency and/or energy consumption has yet to be devised. Therefore, throughout this part of the thesis, we will assume virtual cut-through switching [KK79], whose analysis we present in Chapter 9.

As opposed to deterministic routing, Dumitraş and Mărculescu [DM03] have proposed stochastic communication as a way to deal with permanent and transient faults of network links and nodes. Their method has the advantage of simplicity, low implementation overhead, and high robustness w.r.t. faults. However, their method suffers the disadvantages of non-deterministic routing. Thus, the selection of links and of the number of redundant copies to be sent on the links is stochastically done at runtime by the network routers. Therefore, the transmission latency is unpredictable and, hence, it cannot be guaranteed. More importantly, stochastic communication is very wasteful in terms of energy [Man04].

Pirretti et al. [PLB+04] report significant energy savings relative to Dumitraş' and Mărculescu's approach, while still keeping the low implementation overhead of non-deterministic routing. An incoming packet is forwarded to exactly one outgoing link. This link is randomly chosen according to pre-assigned probabilities that depend on the message source and destination. However, due to the stochastic character of transmission paths and link congestion, neither Dumitraş and Mărculescu, nor Pirretti et al. can provide guarantees on the transmission latency.

As opposed to Dumitraş and Mărculescu and Pirretti et al., who address the problem of reliable communication at system-level, Bertozzi et al. [BBD02] address the problem at on-chip bus level. Bertozzi's approach is based on low-swing signals carrying data encoded with error resilient codes. They analyse the trade-off between consumed energy, transmission latency and error codes, while considering the energy and the chip area of the encoders/decoders. While Bertozzi et al. address the problem at link level, in this chapter we address the problem at application level, considering time-constrained multi-hop transmission of messages sharing the links of an NoC.

Several researchers addressed the problem of dimensioning of the buffers of the on-chip communication infrastructure. Saastamoinen et al. [SAN03] study the properties of on-chip buffers, report gate-area estimates and analyse the buffer utilisation. Chandra et al. [CXSP04] analyse the effect of increasing buffer size on interconnect throughput. However, they use a single source, single sink scenario.

An approach for buffer allocation on NoC is given by Hu and Mărculescu [HM04a]. They consider a design scenario in which an NoC is custom designed for a particular application. Hu and Mărculescu propose a method to distribute a given buffer space budget over the network switches. The algorithm is based on a buffer space demand analysis that relies on given Poisson traffic patterns of the application. Therefore, their approach cannot provide application latency guarantees.

7.3 Highlights of Our Approach

In Chapter 9, we address all of the three stringent problems identified in Section 7.1: link reliability, latency, and energy consumption. We propose a solution for the following problem: Given an NoC architecture with a failure probability for its network links and given an application with required message arrival probabilities and imposed deadlines, find a mapping of messages to network links such that the

imposed message arrival probability and deadline constraints are satisfied at reduced energy costs.

Our approach differs from the approaches of Dumitraş and Mărculescu [DM03] and of Pirretti et al. [PLB+04] in the sense that we *deterministically* select *at design time* the links to be used by each message and the number of copies to be sent on each link. Thus, we are able to guarantee not only message arrival probabilities, but also worst-case message arrival times. In order to cope with the unreliability of on-chip network links, we propose a way to combine spatially and temporally redundant message transmission. Our approach to communication energy reduction is to minimise the application latency at almost no energy overhead by intelligently mapping the redundant message copies to network links. The resulting time slack can be exploited for energy minimisation by means of voltage reduction on network nodes and links.

While the work presented in Chapter 9 tackles the problem of energy-efficient communication, it leaves a potential for further energy and cost savings unexploited. A key factor to the energy and cost-efficiency of applications implemented on NoC is the synthesis of the communication such that buffer needs are kept low.

A poor synthesis of the communication may lead to a high degree of destination contention at ingress buffers of network switches. Undesirable consequences of this contention include long latency and an increased energy consumption due to repeated reads from the buffers [YBD02]. Moreover, a high degree of destination contention runs the risk of buffer overflow and consequently packet drop with significant impact on the throughput [KJS+02]. Even in the presence of a back pressure mechanism, which would prevent packet drops, the communication latency would be severely affected by the packet contention [HM04a]. Thus, in Chapter 10, we concentrate on the buffer space aware communication mapping and packet release timing for applications implemented on NoC.

We focus on two design scenarios, namely the custom design of application-specific NoCs and the implementation of applications on general-purpose NoCs. In the former, the size and distribution of communication buffers can be tailored to precisely fit the application demands. Thus, synthesizing the communication in an intelligent manner could significantly reduce the total need of buffering. In this scenario, the optimisation objective for the communication synthesis approach that we propose is the minimisation of the overall communication buffer space.

In the second design scenario, we assume that an application has to be implemented on a given NoC, with fixed capacity for each buffer. Thus, the challenge consists in mapping the data packets such that no buffer overflow occurs. In both scenarios, it has to be guaranteed that the worst-case task response times are less than the given deadlines, and that the message arrival probability is equal or above an imposed threshold.

Our approach relies on an *analysis* of both timing behaviour and communication buffer space demand at each buffer *in the worst case*. Thus, in both design scenarios, if a solution to the communication synthesis problem is found, we are able to guarantee worst-case timing behaviour and worst-case buffer space demand, which means that no buffer overflows/packet drops occur.

Our approach differs in several aspects from the approach of Hu and Mărculescu [HM04a]. First, in addition to buffer allocation, we perform off-line packet routing under timeliness and buffer capacity constraints. Second, we are able to *guarantee* the application latency and that no packets are dropped due to buffer overflows at the switches. Third, we propose a complementary technique that can be independently deployed for the minimisation of the buffer space demand. This technique consists of delaying the release of packets in order to minimise destination contention at the buffers. The method is sometimes referred to as traffic shaping [RE02].

The next chapter introduces the system model we use throughout this part of the thesis, while Chapter 9 presents our approach to communication mapping for low energy and Chapter 10 describes our approach to buffer space demand minimisation.

Chapter 8

System Modelling

This chapter presents the system model used throughout this part of the thesis.

8.1 Hardware Model

We describe the system model and introduce the notations based on the example in Figure 8.1. The hardware platform consists of a 2D array of $W \times H$ cores, depicted as squares in the figure, where W and H denote the number of columns and rows of the array respectively. The cores are denoted with $P_{x,y}$, where x is the 0-based column index and y is the 0-based row index of the core in the array. The inter-core communication infrastructure consists of a 2D mesh network. The small circles in Figure 8.1 depict the switches, denoted $S_{x,y}$, $0 \leq x < W$, $0 \leq y < H$. Core $P_{x,y}$ is connected to switch $S_{x,y}$, $\forall 0 \leq x < W, 0 \leq y < H$. The thick lines connecting the switches denote the communication links. Each switch, except those on the borders of the 2D mesh, contains five input buffers: one for the link connecting the switch to the core with the same index as the switch, and the rest corresponding to the links conveying traffic from the four neighbouring switches.

The link connecting switch $S_{x,y}$ to switch $S_{x,y+1}$ is denoted with $L_{x,y,N}$ while the link connecting switch $S_{x,y+1}$ to switch $S_{x,y}$ is denoted with $L_{x,y+1,S}$. The link connecting switch $S_{x,y}$ to switch $S_{x+1,y}$ is de-

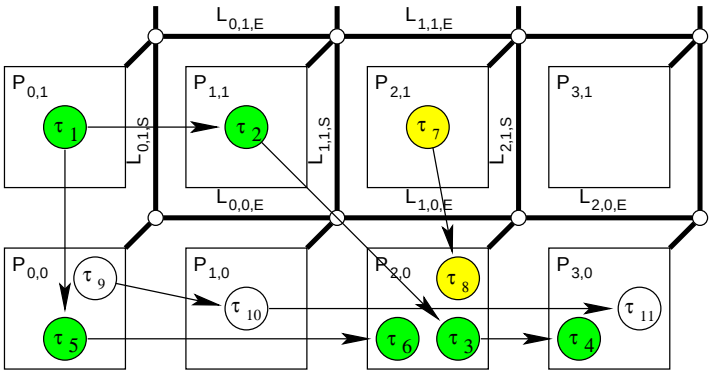


Figure 8.1: Application example

noted with $L_{x,y,E}$ while the link connecting switch $S_{x+1,y}$ to switch $S_{x,y}$ is denoted with $L_{x+1,y,W}$. Each link is characterised by the time and energy it needs to transmit a bit of information.

8.2 Application Model

The application model is similar to the one described in Section 3.2. It diverges from the application model introduced in Section 3.2 in the following aspects:

- Tasks belonging to the same task graph have the same period ($\pi_a = \pi_b$ if $\tau_a, \tau_b \in V_i$ and $(V_i, E_i \subset V_i \times V_i)$ is a task graph.)
- The task execution time probability density functions are unknown. For each task $\tau_i \in T$, we know only the upper and lower bounds on its execution time, $WCET_i$ and $BCET_i$ respectively.
- There are no limits on the maximum number of task graph instantiations that may be active in the system at the same time.
- Tasks are scheduled according to a fixed-priority preemptive scheduling policy.

8.3 Communication Model

Communication between pairs of tasks mapped on different cores is performed by message passing. Their transmission on network links is done packet-wise, i.e. the message is chopped into packets, which are sent on links and reassembled at the destination core. Messages are characterised by their priority, length (number of bits), and the size of the packets they are chopped into.

If an output link of a switch is busy sending a packet while another packet arrives at the switch and demands forwarding on the busy link, the newly arrived packet is stored in the input buffer corresponding to the input link on which it arrived. When the output link becomes available, the switch picks the highest priority packet that demands forwarding on the output link. If an output link of a switch is not busy while a packet arrives at the switch and demands forwarding on the busy link, then the packet is forwarded immediately, without buffering. This scheme is called virtual cut-through routing [KK79].

Packet transmission on a link is modelled as a task, called communication task. The worst-case execution time of a communication task is given by the packet length divided by the link bandwidth. The execution of communication tasks is non-preemptible.

8.4 Fault Model

Communication links may temporarily malfunction, with a given probability. If a data packet is sent on the link during the time the link is in the failed state, the data is scrambled. We assume that the switches have the ability to detect if an incoming packet is scrambled. Scrambled packets are dropped as soon as they are detected and are not forwarded further. Several copies of the same packet may be sent on the network links. In order for a message to be successfully received, at the destination core, at least one copy of every packet of the message has

to reach the destination core unscrambled. Otherwise, the message is said to be lost.

We define the message arrival probability of the message $\tau_i \rightarrow \tau_j$ as the long term ratio $MAP_{i,j} = \lim_{t\to\infty} \frac{S_{i,j}(t)}{\lfloor t/\pi_i \rfloor}$, where $S_{i,j}$ is the number of messages between tasks τ_i and τ_j that are successfully received at the destination in the time interval $[0, t)$, and π_i denotes the period of the sender task. For each pair of communicating tasks $\tau_i \rightarrow \tau_j$, the designer may require lower bounds $B_{i,j}$ on the ratio of messages that are received unscrambled at the destination.

Let us assume that switches have the capability to detect erroneous (scrambled) packets, but they do not have the capacity to correct these errors. We let α denote the probability of a packet to traverse a network link unscrambled. A strategy to satisfy the constraints on the message arrival probability ($MAP_{i,j} \geq B_{i,j}$) is to make use of spatially and/or temporally redundant packet transmission, i.e. several copies of the same packet are simultaneously transmitted on different paths and/or they are resent several times on the same path. This strategy is discussed in Chapter 9.

An alternative strategy to cope with transient faults on the network links is to add redundant bits to the packets for error correction. Additionally, extra circuitry has to be deployed at the switches such that they can correct some of the erroneous packets. In this case, let α denote the probability of a packet successfully traversing a link, which means that

- the packet traverses the network link unscrambled, *or*
- the packet is scrambled during its transmission along the link, but the error correction circuitry at the end of the link is able to correct the scrambled bits.

Note that the two strategies are orthogonal, in the sense that the redundant transmission can be deployed even when error correction capability is present in the network. In this case, redundant transmission attempts to cope with the errors that are detected but cannot be corrected by the correction circuitry. An analysis of the trade-off between the two strategies is beyond the scope of this thesis.

In the sequel, we will make use of α, the probability of a packet to successfully traverse a link, abstracting away whether the successful transmission is due to error correction or not.

8.5 Message Communication Support

In order to satisfy message arrival probabilities imposed by the designer, temporally and/or spatially redundant communication is deployed. We introduce the notion of *communication supports (CS)* for defining the mapping of redundant messages to network links. For this purpose, we use the example in Figure 8.1. A possible mapping of messages to network links is depicted in Figure 8.2. The directed lines depicted parallel to a particular link denote that the message represented by the directed line is mapped on that link. Thus, message $\tau_1 \rightarrow \tau_2$ is conveyed by link $L_{0,1,E}$, message $\tau_1 \rightarrow \tau_5$ by link $L_{0,1,S}$, message $\tau_7 \rightarrow \tau_8$ by link $L_{2,1,S}$, message $\tau_9 \rightarrow \tau_{10}$ by link $L_{0,0,E}$, message $\tau_5 \rightarrow \tau_6$ by links $L_{0,0,E}$ and $L_{1,0,E}$, message $\tau_{10} \rightarrow \tau_{11}$ by links $L_{1,0,E}$ and $L_{2,0,E}$.

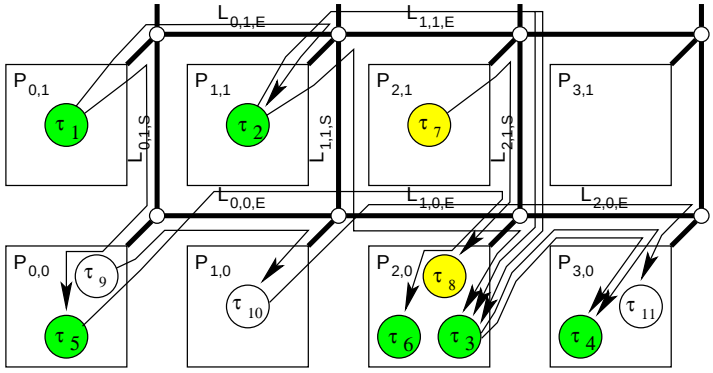


Figure 8.2: Message mapping for the application in Figure 8.1

Of particular interest are messages $\tau_3 \rightarrow \tau_4$ and $\tau_2 \rightarrow \tau_3$. *Two* identical copies of the former are sent on the same link, namely link $L_{2,0,E}$, as indicated by the double arrow between task τ_3 and τ_4 in the figure. Therefore, the transmission of message $\tau_3 \rightarrow \tau_4$ is temporally redundant. A more complex case is exemplified by message $\tau_2 \rightarrow \tau_3$. Identical copies of the message take different routes. Therefore, the transmission of message $\tau_2 \rightarrow \tau_3$ is spatially redundant. One copy is conveyed by links $L_{1,1,E}$ and $L_{2,1,S}$, while the second copy is conveyed by links $L_{1,1,S}$ and $L_{1,0,E}$. Moreover, the copy travelling along the first route is in its turn replicated once it reaches switch $S_{2,3}$ and sent twice on link $L_{2,1,S}$, as shown by the double arrow in the figure.

In general, the mapping of the communication between two tasks $\tau_i \rightarrow \tau_j$ can be formalised as a set of tuples $CS_{i,j} = \{(L, n) : L \text{ is a link}, n \in \mathbb{N}\}$, where n indicates the number of copies of the same message that are conveyed by the corresponding link L. We will call the set $CS_{i,j}$ the *communication support* (CS) of $\tau_i \rightarrow \tau_j$.

Let $\mathcal{M} \subset T \times T$ be the set of all pairs of communicating tasks that are mapped on different cores ($(\tau_a, \tau_b) \in \mathcal{M}$ iff $\exists \Gamma_i = (V_i, E_i \subset V_i \times V_i)$ such that $(\tau_a, \tau_b) \in E_i$ and $Map(\tau_i) \neq Map(\tau_j)$.) A *communication mapping*, denoted $\mathcal{CM}$, is a function defined on $\mathcal{M}$ that maps each pair of communicating tasks to one communication support.

In our example, the communication mapping is the following: $\tau_1 \rightarrow \tau_2$ is mapped on $CS_{1,2} = \{(L_{0,1,E}, 1)\}$, $\tau_1 \rightarrow \tau_5$ on $CS_{1,5} = \{(L_{0,1,S}, 1)\}$, $\tau_7 \rightarrow \tau_8$ on $CS_{7,8} = \{(L_{2,1,S}, 1)\}$, $\tau_9 \rightarrow \tau_{10}$ on $CS_{9,10} = \{(L_{0,0,E}, 1)\}$, $\tau_{10} \rightarrow \tau_{11}$ on $CS_{10,11} = \{(L_{1,0,E}, 1), (L_{2,0,E}, 1)\}$, $\tau_5 \rightarrow \tau_6$ on $CS_{5,6} = \{(L_{0,0,E}, 1), (L_{1,0,E}, 1)\}$, $\tau_3 \rightarrow \tau_4$ on $CS_{3,4} = \{(L_{2,0,E}, 2)\}$, and $\tau_2 \rightarrow \tau_3$ on $CS_{2,3} = \{(L_{1,1,E}, 1), (L_{2,1,S}, 2), (L_{1,1,S}, 1), (L_{1,0,E}, 1)\}$.

Two properties of a communication support are of interest:

- The arrival probability of a message that is mapped on that communication support, called message arrival probability of the CS, denoted MAP, and
- The expected energy consumed by the transmission of the message on that support, called expected communication energy of the CS, denoted ECE.

The values of MAP and ECE can be computed by means of simple probability theory. We will illustrate their computation using the CS supporting message $\tau_2 \rightarrow \tau_3$, $CS_{2,3}$. For simplicity, in this example we

assume that the message consists of a single packet and the energy consumed by the transmission of the packet on any link is 1.

The MAP of $CS_{2,3}$ is given by $\mathsf{P}(V \cup W)$, where V is the event that the copy sent along the path $L_{1,1,S} \rightarrow L_{1,0,E}$ successfully reaches core $P_{2,0}$, and W is the event that the copy sent along the path $L_{1,1,E} \rightarrow L_{2,1,S}$ successfully reaches core $P_{2,0}$.

$$\mathsf{P}(V) = \alpha^2.$$

The probability that both temporally redundant copies that are sent on link $L_{2,1,S}$ get scrambled is $(1-\alpha)^2$. Thus, the probability that the packet successfully reaches core $P_{2,0}$ if sent on path $L_{1,1,E} \rightarrow L_{2,1,S}$ is

$$\mathsf{P}(W) = \alpha \cdot (1 - (1-\alpha)^2).$$

Thus, the MAP of $CS_{2,3}$ is

$$\begin{aligned}\mathsf{P}(V \cup W) &= \mathsf{P}(V) + \mathsf{P}(W) - \mathsf{P}(V \cap W) = \\ &= \alpha^2 + \alpha \cdot (1 - (1-\alpha)^2) - \alpha^3 \cdot (1 - (1-\alpha)^2).\end{aligned}$$

The expected communication energy is the expected number of sent bits multiplied by the average energy per bit. The energy per bit, denoted E_{bit}, can be computed as shown by Ye et al. [YBD02].

The ECE of $CS_{2,3}$ is proportional to

$$E[Sent_{S_{1,1}}] + E[Sent_{S_{2,1}}] + E[Sent_{S_{1,0}}],$$

where $Sent_S$ denotes the number of packets sent from switch S and $E[Sent_S]$ denotes its expected value.

$$E[Sent_S] = E[Sent_S | R_S] \cdot \mathsf{P}(R_S),$$

where R_S is the event that at least one copy of the packet successfully reaches switch S, end $E[Sent_S|R_S]$ is the number of copies of the packet that are forwarded from switch S given that at least one copy successfully reaches switch S. Hence,

$$ECE_{2,3} \sim 2 + 2 \cdot \alpha + 1 \cdot \alpha = 2 + 3\alpha.$$

The proportionality constant is $E_{bit} \cdot b$, where E_{bit} is the energy per bit and b is the number of bits of the packet.

Chapter 9

Energy and Fault-Aware Time-Constrained Communication Synthesis for NoC

In this chapter we present an approach for determining the communication support for each message such that the communication energy is minimised, the deadlines are met and the message arrival probability is higher than imposed lower bounds. The approach is based on constructing a set of promising communication support candidates for each message. Then, the space of communication support candidates is explored in order to find a communication support for each message such that the task response times are minimised. In the last step of our approach, the resulted execution time slack can be exploited by means of voltage and/or frequency scaling in order to reduce the communication energy.

9.1 Problem Formulation

This section gives the formulation of the problem that we solve in this chapter.

9.1.1 Input

The input to the problem consists of:

- The hardware model, i.e. the size of the NoC, and, for each link, the energy-per-bit, the bandwidth, and the probability of a packet to be successfully conveyed by the link; and
- The application model, i.e. the set of task graphs Γ, the set of task and task graph deadlines Δ_T and Δ_Γ respectively, the mapping of tasks to cores, the set of task periods Π_T, the best-case and worst-case execution times of all tasks on the cores on which they are mapped, the task priorities and the amounts of data to be transmitted between communicating tasks;

(1) **for each** pair of communicating tasks $\tau_i \rightarrow \tau_j$
(2) *find* a set of candidate CSs that satisfies $MAP_{i,j} \geq B_{i,j}$ (Section 9.3)
(3) **end for**
(4) $(sol, min\_cost) = explore$ the space of candidate CSs (Section 9.5)
(5) using *response time calculation* (Section 9.4) for driving the exploration
(6) **if** $min\_cost = \infty$ **then**
(7) **return** "no solution"
(8) **else**
(9) sol' =voltage_freq_selection(sol) (according to [ASE+04])
(10) **return** sol'
(11) **end if**

Figure 9.1: Approach outline

- The communication model, i.e. the packet size and message priority for each message (alternatively, our approach can automatically assign message priorities according to the message criticality);
- The lower bounds $B_{i,j}$ imposed on the message arrival probability $MAP_{i,j}$, which is the expected fraction of successfully transmitted messages, for each pair of communicating tasks $\tau_i \rightarrow \tau_j$.

9.1.2 Output

The output of the problem consists of the communication mapping $\mathcal{CM}$, such that the total communication energy is minimised.

9.1.3 Constraints

The application has to satisfy the following constraints:

- For each pair of communicating tasks $\tau_i \rightarrow \tau_j$, such that tasks τ_i and τ_j are mapped on different cores, the message arrival probability $MAP_{i,j}$ is greater than or equal to the imposed lower bound $B_{i,j}$ ($MAP_{i,j} \geq B_{i,j}, \forall \tau_i, \tau_j \in T : \tau_i \in {}^\circ\tau_j \wedge Map(\tau_i) \neq Map(\tau_j)$).
- All deadlines are met.

9.2 Approach Outline

The outline of our approach to solve the problem is shown in Figure 9.1. First, for each pair of communicating tasks (message), we find a set of candidate communication supports (line 2, see Section 9.3), such that the lower bound constraint on the message arrival probability is satisfied. Second, the space of candidate communication supports is explored in order to find sol, the selection of communication supports that result in the minimum cost min_cost (line 4).[1] The worst-case response

[1]See Section 9.5 for a precise definition of the cost of a solution. Intuitively, a low cost corresponds to a solution characterised by large time slack (long intervals between the finishing time of a task and its deadline.)

time of each explored solution is determined by the response time calculation function that drives the design space exploration (line 5, see Section 9.4). If no solutions are found that satisfy the response time constraints ($min\_cost = \infty$), the application is deemed impossible to implement with the given resources (line 7). Otherwise, the solution with the minimum cost among the found solutions is selected. Voltage selection is performed on the selected solution in order to decrease the overall system energy consumption (line 9), and the modified solution is returned (line 10).

The next section discusses the construction of the set of candidate communication supports for an arbitrary pair of communicating tasks. Section 9.4 describes how the response time calculation is performed, while Section 9.5 outlines how the preferred communication supports representing the final solution are selected.

9.3 Communication Support Candidates

This section describes how to construct a set of candidate communication supports for a pair of communicating tasks. First we introduce the notions of path, coverage, and spatial, temporal, and general redundancy degree of a CS.

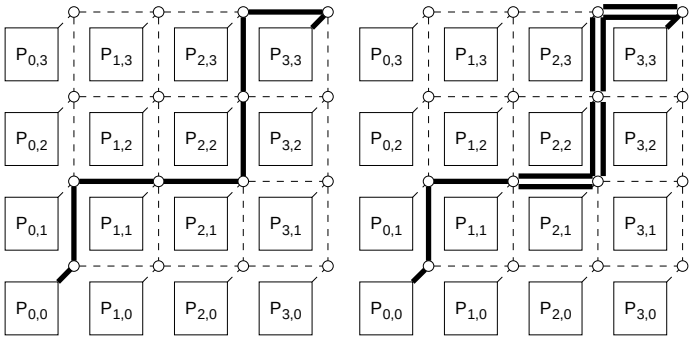
A path of length n connecting the switch corresponding to a source core to a switch corresponding to a destination core is an ordered sequence of n links, such that the end point of the i^{th} link in the sequence coincides with the start point of the $i + 1^{th}$ link, $\forall 1 \leq i < n$, and the start point of the first link is the source switch and the end point of the last link is the destination switch. We consider only loop-free paths. A path *belongs* to a CS if all its links belong to the CS. A link of a CS is *covered* by a path if it belongs to the path.

The spatial redundancy degree (SRD) of a CS is given by the minimum number of distinct paths belonging to the CS that cover all the links of the CS. For example, the CSs depicted in Figures 9.2(a) and 9.2(b) both have a SRD of 1, as they contain only one path, namely path ($L_{0,0,N}$, $L_{0,1,E}$, $L_{1,1,E}$, $L_{2,1,N}$, $L_{2,2,N}$, $L_{2,3,E}$). The CS shown in Figure 9.2(c) has spatial redundancy degree 2, as at least two paths are necessary in order to cover links $L_{1,1,N}$ and $L_{1,1,E}$, for example paths ($L_{0,0,N}$, $L_{0,1,E}$, $L_{1,1,E}$, $L_{2,1,N}$, $L_{2,2,N}$, $L_{2,3,E}$) and ($L_{0,0,N}$, $L_{0,1,E}$, $L_{1,1,N}$, $L_{1,2,E}$, $L_{2,2,N}$, $L_{2,3,E}$).

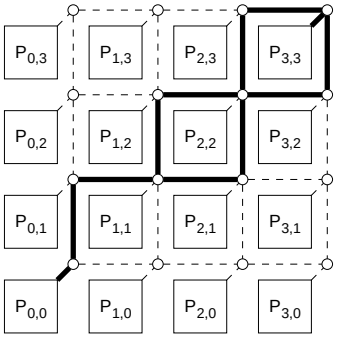
The temporal redundancy degree (TRD) of a link is given by the number of redundant copies to be sent on the link. The TRD of a CS is given by the maximum TRD of its links. For example, the TRD of the CS shown in Figure 9.2(b) is 2 as two redundant copies are sent on links $L_{1,1,E}$, $L_{2,1,N}$, $L_{2,2,N}$, and $L_{2,3,E}$. The TRD of the CSs shown in Figures 9.2(a) and 9.2(c) is 1.

The general redundancy degree (GRD) of a CS is given by the sum of temporal redundancy degrees of all its links. For example, the GRD of the CS shown in Figure 9.2(a) is 6, the GRD of the CSs shown in Figures 9.2(b) and 9.2(c) is 10.

It is important to use CSs of minimal GRD because the expected communication energy (ECE) of a message is strongly dependent on the GRD of the CS supporting it. To illustrate this, we constructed all CSs of SRD 2 and GRD 10–13 for a message sent from the lower-left core to the upper-right core of a 4×4 NoC. We also constructed



(a) (b)



(c)

Figure 9.2: Communication supports

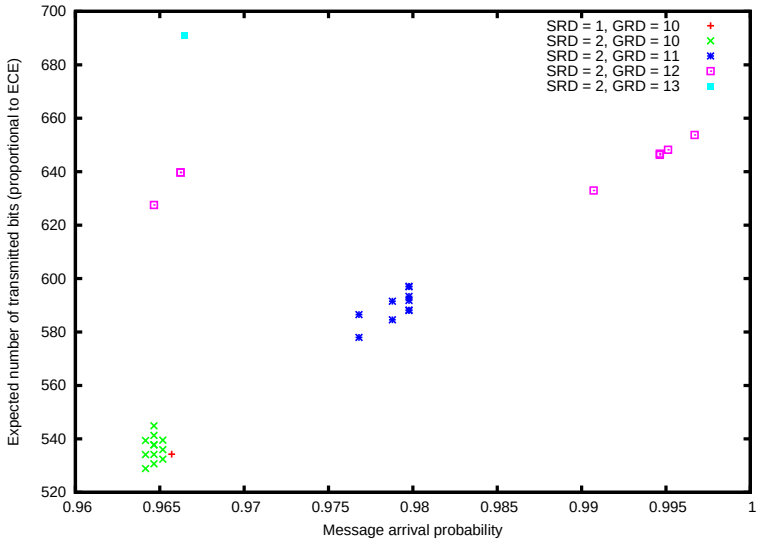


Figure 9.3: Energy-efficiency of CSs of SRD 1 and 2

(1) **for each** pair of communicating tasks $\tau_i \rightarrow \tau_j$
(2) Determine N_1 and N_2, the minimum general redundancy degrees of CSs of SRD 1 and 2 respectively, such that the MAP constraint on $\tau_i \rightarrow \tau_j$ is satisfied
(3) Add all CSs with SRD 1 and with GRD N_1 and all CSs with SRD 2 and with GRD N_2 to the set of CS candidates of $\tau_i \rightarrow \tau_j$
(4) **end for**

Figure 9.4: Construction of candidate CS set

all CSs of SRD 1 and GRD 10. For each of the constructed CS, we computed their MAP and ECE. In Figure 9.3, we plotted all resulting (MAP, ECE) pairs. Note that several different CSs may have the same MAP and ECE and therefore one dot in the figure may correspond to many CSs. We observe that the ECE of CSs of the same GRD do not differ significantly among them, while the ECE difference may account to more than 10% for CSs of different GRD.

The algorithm for the candidate set construction proceeds as shown in Figure 9.4. Candidate CSs with SRD of only 1 and 2 are used. The justification for this choice is given later in the section.

We illustrate how to find the minimal GRD for a message based on the example depicted in Figure 9.2. We consider a 4×4 NoC, and a message sent from core $P_{0,0}$ to core $P_{3,3}$. The message consists of just one packet, the probability that the packet successfully traverses any of the links is $\alpha = 0.99$, and the imposed lower bound on the MAP is $B = 0.975$.

We look first at CSs with SRD of 1, i.e. consisting of a single path. We consider only shortest paths, that is of length 6. Obviously, a lower bound on GRD is 6. If we assign just one copy per link, the message arrival probability would be $\alpha^6 \approx 0.941 < 0.975 = B$. We try with a GRD of 7, and regardless to which of the 6 links we assign the redundant copy, we get a MAP of $\alpha^5 \cdot (1 - (1 - \alpha)^2) \approx 0.95 < 0.975 = B$. Hence, we are forced to increase the GRD once more. We observe that there are 5 links left with a TRD of 1. The probability to traverse them is $\alpha^5 \approx 0.95$, less than the required lower bound. Therefore it is useless to assign one more redundant copy to the link that now has a TRD of 2 because anyway the resulting MAP would not exceed α^5. Thus, the new redundant copy has to be assigned to a different link of the CS of GRD 8. In this case, we get a MAP of $\alpha^4 \cdot (1 - (1 - \alpha)^2)^2 \approx 0.96$, still less than the required bound. We continue the procedure of increasing the GRD and distributing the redundant copies to different links until we satisfy the MAP constraint. In our example, this happens after adding 4 redundant copies ($MAP = \alpha^2 \cdot (1 - (1 - \alpha^2))^4 \approx 0.9797$). The resulting CS of SRD 1 and GRD 10 is shown in Figure 9.2(b), where the double lines represent links that convey two copies of the same packet. Thus, the minimal GRD for CSs of SRD 1 is $N_1 = 10$. There are 20 distinct paths between core $P_{0,0}$ and core $P_{3,3}$ and there are 15 ways of distributing the 4 redundant copies to each path. Thus, $15 \cdot 20 = 300$ distinct candidate CSs of SRD 1 and GRD 10 can be constructed for the considered message. They all have the same message arrival probability, but different expected communication energies. The ECEs among them vary 1.61%.

Similarly, we obtain N_2, the minimal GRD for CSs of SRD 2. In this case, it can be mathematically shown that larger message arrival probabilities can be obtained with the same GRD if the two paths of the CS intersect as often as possible and the distances between the intersection points are as short as possible [Man04]. Intuitively, intersection points are important because even if a copy is lost on one incoming path, the arrival of another copy will trigger a regeneration of two packets in the core where the two paths intersect. The closer to each other the intersection points are, the shorter the packet transmission time between the two points is. Thus, the probability to lose a message between the two intersection points is lower. Therefore, in order to obtain N_2, we will consider CSs with many intersection points that are close to each other. For our example, the lowest GRD that lets the CS satisfy the MAP constraint is $N_2 = 10$ ($MAP = \alpha^6 \cdot (2-\alpha^2)^2 \approx 0.9793$). This CS is shown in Figure 9.2(c). The minimum number of needed redundant copies in order to satisfy the MAP constraint is strongly dependent on α and the imposed lower bound on the MAP, and only weakly dependent on the geometric configuration of the CS. Therefore, typically $N_2 = N_1$ or it is very close to N_1.

In conclusion, N_1 and N_2 are obtained by progressively increasing the GRD until the CS satisfies the MAP constraint. The redundant copies must be uniformly distributed over the links of the CS. Additionally, in the case of CSs with SRD 2, when increasing the GRD, links should be added to the CS such that many path intersection points are obtained and that they are close to each other.

The following reasoning lies behind the decision to use CSs with SRD of only 1 and 2. First, we give the motivation for using CSs with SRD larger than 1. While, given a GRD of N, it is possible to obtain the maximum achievable message arrival probability with CSs of SRD 1, concurrent transmission of redundant message copies would be impossible if we used CSs with SRD of only 1. This could severely affect the message latency or, even worse, lead to link overload. CSs with SRD 2 are only marginally more energy hungry, as can be seen from the cluster of points in the lower-left corner of Figure 9.3. Usually, the same MAP can be obtained by a CS of SRD 2 with only 1–2% more energy than a CS of SRD 1.

While the previous consideration supports the use of CSs with SRD greater than 1, there is no reason to go with the SRD beyond 2. Because of the two-dimensional structure of the NoC, there are at most 2 different links that belong to the shortest paths between the source and the destination and whose start points coincide with the source core. Thus, if a CS consisted only of the shortest paths, the message transmission would be vulnerable to a double fault of the two initial links. Therefore, CSs with SRD greater than 2, while consuming more energy for communication, would still be vulnerable to a double fault on the initial links and hence can only marginally improve the MAP. If we did not restrict the CS to the shortest paths, while overcoming the limitation on the MAP, we would consume extra energy because of the longer paths. At the same time, latency would be negatively affected. Thus, for two-dimensional NoC, we consider CSs of SRD of only 1 and 2.

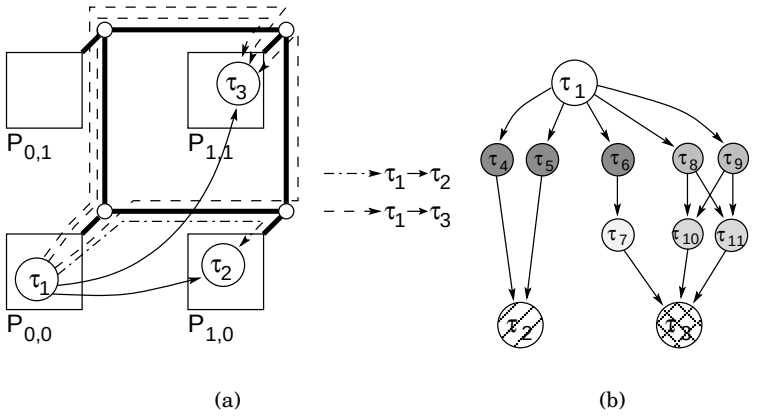


Figure 9.5: Application modelling for response time analysis

9.4 Response Time Calculation

In order to guarantee that tasks meet their deadlines, in case no message is lost, response times have to be determined in the worst case.

Let us consider the example depicted in Figure 9.5(a). Solid lines depict data dependencies among the tasks, while the dotted lines show the actual communication mapping to the on-chip links. The two CSs are $CS_{1,2} = \{(L_{0,0,E}, 1)\}$ and $CS_{1,3} = \{(L_{0,0,E}, 1), (L_{1,0,N}, 1), (L_{0,0,N}, 2), (L_{0,1,E}, 2)\}$. Packet sizes are such that message $\tau_1 \rightarrow \tau_2$ is chopped into 2 packets, while message $\tau_1 \rightarrow \tau_3$ fits into a single packet.

Based on the application graph, its mapping and the communication supports, we construct a task graph as shown in Figure 9.5(b). Each link L is regarded as a processor P_L, and each packet transmission on link L is regarded as a non-preemptive task executed on processor P_L. The shadings of the circles denote the processors (links) on which the tasks (packets) are mapped. Tasks τ_4 and τ_5 represent the first and the second packet of the message $\tau_1 \rightarrow \tau_2$. They are both dependent on task τ_1, as the two packets are generated when task τ_1 completes its execution, while task τ_2 is dependent on both task τ_4 and τ_5 as it can start only after it has received the entire message, i.e. both packets, from task τ_1. Both tasks τ_4 and τ_5 are mapped on the "processor" corresponding to the link $L_{0,0,E}$. Task τ_6 represents the packet of the message $\tau_1 \rightarrow \tau_3$ that is sent on link $L_{0,0,E}$ and task τ_7 represents the same packet once it reaches link $L_{1,0,N}$. Tasks τ_8 and τ_9 are the two copies of the packet of the message $\tau_1 \rightarrow \tau_3$ that are sent on link $L_{0,0,N}$.

We are interested in the worst-case scenario w.r.t. response times. In the worst case, all copies of packets get scrambled except the latest packet. Therefore, the copies to be sent by a core on its outgoing links have to wait until the last of the copies arriving on incoming links of the core has reached the core. For example, tasks τ_{10} and τ_{11}, modelling the two copies of the message $\tau_1 \rightarrow \tau_3$ that are sent on the link $L_{0,1,E}$, depend on both τ_8 and τ_9, the two copies on link $L_{0,0,N}$. Also, task τ_3 depends on all three copies, τ_7, arriving on link $L_{1,0,N}$, and τ_{10} and τ_{11}, arriving on link $L_{0,1,E}$.

The modified model, as shown in Figure 9.5(b), is analysed using the dynamic offset based schedulability analysis proposed by Palencia

and Harbour [PG98]. The analysis calculates the worst-case response times and jitters for all tasks.

9.5 Selection of Communication Supports

As shown in Section 9.3 (see also line 2 in Figure 9.1), we have determined the most promising (low energy, low number of messages) set of CSs for each transmitted message in the application. All those CSs guarantee the requested MAP. As the next step of our approach (line 4 in Figure 9.1) we have to select one particular CS for each message, such that the solution cost is minimised, which corresponds to maximising the smallest time slack. The response time for each candidate solution is calculated as outlined in Section 9.4 (line 5 in Figure 9.1).

The design space is explored with the Tabu Search based heuristic. The basic principles of Tabu Search have been described in Section 6.4.1. The design space is the Cartesian product of the sets of CS candidates for each message (constructed as shown in Section 9.3.) Because all CS candidates guarantee the requested MAP, all points in the solution space satisfy the MAP constraint of the problem (Section 9.1.3). A point in the design space is an assignment of communication supports to messages (see Section 8.5). A move means picking one pair of communicating tasks and selecting a new communication support for the message sent between them. In order to select a move, classical Tabu Search explores all solutions that can be reached by one move from the current solution. For each candidate solution, the application response time has to be calculated. Such an approach would be too time consuming for our problem. Therefore, we only explore "promising" moves. Thus,

1. we look at messages with large jitters as they have a higher chance to improve their transmission latency by having assigned a new CS; and
2. for a certain message $\tau_i \rightarrow \tau_j$, we consider only those candidate CSs that would decrease the amount of interference of messages of higher priority than $\tau_i \rightarrow \tau_j$. (By this we remove messages from overloaded links.)

The value of the cost function that drives the response-time minimisation, evaluated for an assignment of communication supports to messages $\mathcal{CS}$, is:

$$cost(\mathcal{CS}) = \begin{cases} \infty & \exists \tau \in T : WCRT_\tau > \delta_\tau \vee \\ & \vee \exists \Gamma_i \in \Gamma : WCRT_{\Gamma_i} > \delta_{\Gamma_i} \\ \max_{\tau \in T} \frac{WCRT_\tau}{\delta_\tau} & \text{otherwise,} \end{cases} \tag{9.1}$$

where T is the set of tasks and $WCRT$ and δ denote worst-case response times and deadlines respectively. The worst-case response time of a task is obtained as shown in Section 9.4.

In the case of the cost function in Eq. (9.1), we make the conservative assumption that voltage and frequency are set system-wide for the whole NoC. This means that, at design time, an optimal voltage and/or frequency is determined for the whole NoC. The determined values for voltage and frequency do not change during the entire execution time of the application. For such a scenario, if we decrease the system-wide

voltage (or frequency), the worst-case response times of all tasks would scale with the same factor. Therefore, in the definition of the cost function (Eq. (9.1)) we use the max operator, as the width of the interval in which the response time is allowed to increase is limited by the smallest slack (largest $\frac{WCRT_{\tau_i}}{\delta_{\tau_i}}$) among the tasks.

In a second scenario, we can assume that voltage and/or frequency may be set core-wise. This means that, at design time, an optimal voltage and/or frequency is calculated for each core. These voltages and frequencies do not change during the whole execution time of the application. The cost function would become

$$cost(\mathcal{CS}) = \begin{cases} \infty & \exists \tau \in T : WCRT_\tau > \delta \vee \\ & \exists \Gamma_i \in \Gamma : WCRT_{\Gamma_i} > \delta_{\Gamma_i} \\ \sum\limits_{p \in P} \max_{\tau \in \mathcal{T}_p} \frac{WCRT_\tau}{\delta_\tau} & \text{otherwise,} \end{cases} \tag{9.2}$$

where p is a core in P, the set of cores of the NoC, and $\mathcal{T}_p$ is the set of tasks mapped on core p.

If we assume that voltage and/or frequency may change for each core during operation, then they may be set task-wise and the cost function becomes

$$cost(\mathcal{CS}) = \begin{cases} \infty & \exists \tau \in T : WCRT_\tau > \delta \vee \\ & \vee \exists \Gamma_i \in \Gamma : WCRT_{\Gamma_i} > \delta_{\Gamma_i} \\ \sum\limits_{\tau \in T} \frac{WCRT_\tau}{\delta_\tau} & \text{otherwise.} \end{cases} \tag{9.3}$$

9.6 Experimental Results

We report on three sets of experiments that we ran in order to assess the quality of our approach.

9.6.1 Latency as a Function of the Number of Tasks

The first set investigates the application latency as a function of the number of tasks. 340 applications of 16 to 80 tasks were randomly generated. The applications are executed by a 4×4 NoC. The probability that a link successfully conveys a data packet is 0.97, and the imposed lower bound on the message arrival probability is 0.99. For each application, we ran our communication mapping tool twice. In the first run, we consider CSs of SRD 1, i.e. packets are retransmitted on the same, unique path. In the second run, we consider CSs of SRD 1 and 2, as described in Section 9.3. Figure 9.6 depicts the averaged results. The approach that uses both spatially and temporally redundant CSs leads to shorter application latencies than the approach that just re-sends on the same path.

9.6.2 Latency as a Function of the Imposed Message Arrival Probability

The second experiment investigates the dependency of latency on the imposed message arrival probability. 20 applications, each of 40 tasks, were randomly generated. We considered the same hardware platform

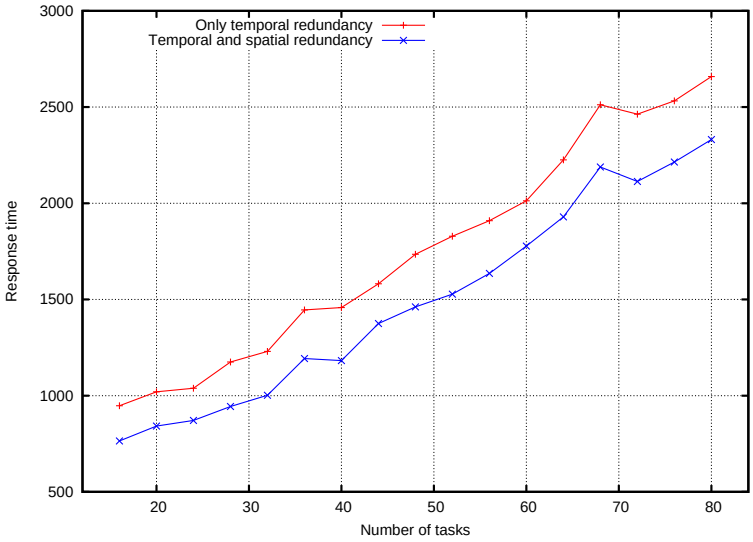


Figure 9.6: Application latency vs number of tasks

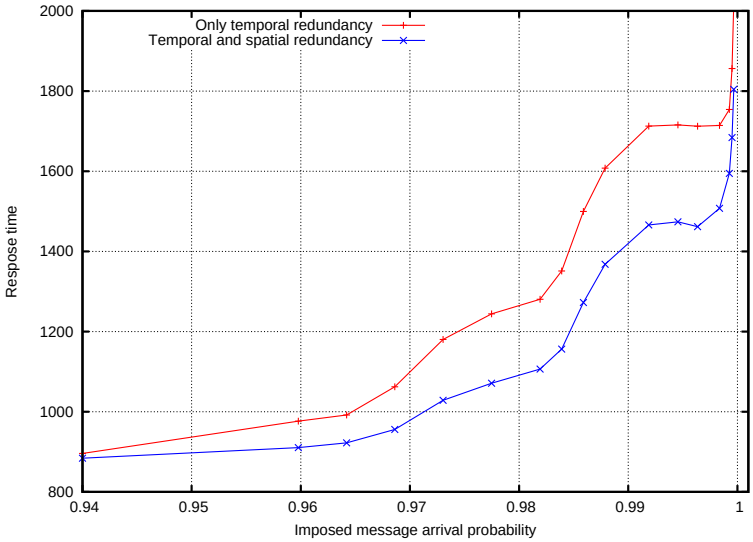


Figure 9.7: Application latency vs bound on MAP

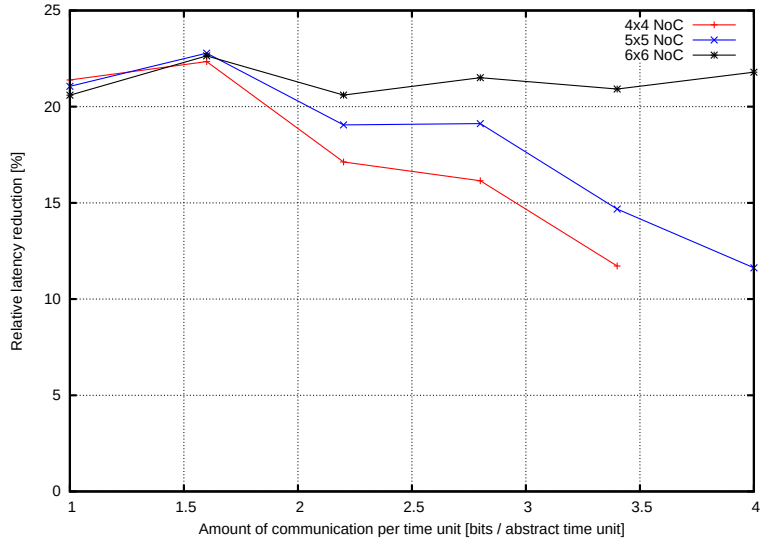


Figure 9.8: Application latency vs NoC size and communication load

as in the first experiment. For each application, we considered 17 different lower bounds on MAP, ranging from 0.94 to 0.9966. The averaged results are shown in Figure 9.7. For low bounds on MAP, such as 0.94, almost no transmission redundancy is required to satisfy the MAP constraint. Therefore, the approach combining spatially and temporally redundant communication fares only marginally better than the approach that uses only temporal redundancy. However, for higher bounds on the MAP, the approach that combines spatially and temporally redundant transmission has the edge. In the case of bounds on the MAP larger than 0.9992, spatial redundancy cannot satisfy the constraint anymore, and therefore the temporally redundant transmission becomes dominant and the approach combining spatial and temporal redundancy does not lead to significant latency reductions anymore.

9.6.3 Latency as a Function of the Size of the NoC and Communication Load

The third experiment has a double purpose. First, it investigates the dependency of latency reduction on the size of the NoC. Second, it investigates latency reduction as a function of the communication load (bits/time unit). 20 applications of 40, 62 and 90 tasks were randomly generated. The applications with 40 tasks run on a 4×4 NoC, those with 62 tasks run on a 5×5 NoC and those with 90 tasks run on a 6×6 NoC. For each application, we considered communication loads of 1–4 bits/time unit. The averaged latency reductions when using the optimal combination of spatial and temporal redundancy, compared to purely temporal redundancy, are depicted in Figure 9.8. We observe that for low communication loads, the latency reduction is similar for all three architectures, around 22%. However, at loads higher than 3.4 the relatively small number of links of the 4×4 NoC get congested and response times grow unboundedly. This, however, is not the case with the larger NoCs. Latency reduction for a load of 4 is 22% for a NoC of 6×6 and 12% for 5×5.

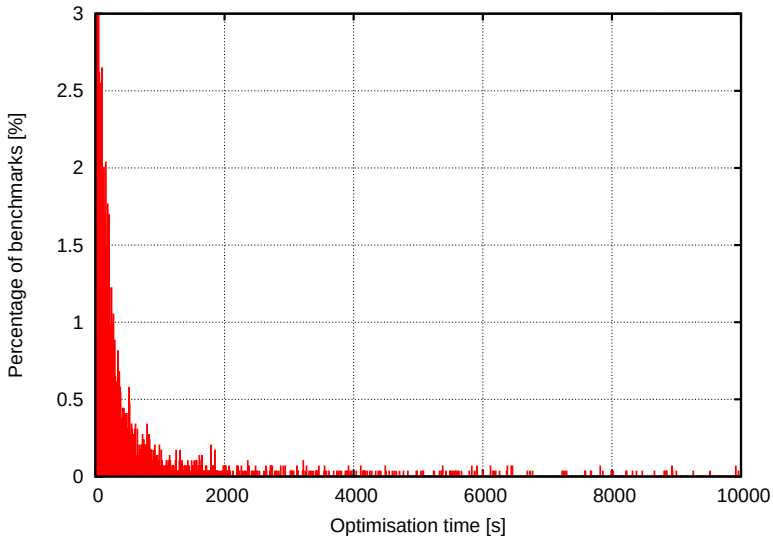


Figure 9.9: Histogram of the optimisation time as measured on an AMD Athlon@1533MHz desktop PC

9.6.4 Optimisation Time

Figure 9.9 depicts the histogram of the optimisation time for all benchmarks that are used in this section, as measured on a desktop PC with an AMD Athlon processor clocked at 1533 MHz. On average, the optimisation time is 912 seconds. We note that the optimisation time for a large majority of benchmarks is smaller than 1000 seconds, while 5.6% of all benchmarks took between 1000 and 2000 seconds to optimise, and the optimisation of 8.7% of benchmarks took longer than 2000 seconds.

9.6.5 Exploiting the Time Slack for Energy Reduction

The presented experiments have shown that, by using an optimal combination of temporal and spatial redundancy for message mapping, significant reduction of latency can be obtained while guaranteeing message arrival probability at the same time. It is important to notice that the latency reduction is obtained without energy penalty, as shown in Section 9.3. This means that for a class of applications using the proposed approach it will be possible to meet the imposed deadlines, which otherwise would not be possible without changing the underlying NoC architecture. However, the proposed approach gives also the opportunity to further reduce the energy consumed by the application. If the obtained application response time is smaller than the imposed one, the resulting slack can be exploited by running the application at reduced voltage. In order to illustrate this, we have performed another set of experiments.

Applications of 16 to 60 tasks running on a 4×4 NoC were randomly generated. For each application we ran our message mapping approach twice, once using CSs with SRD of only 1, and second using CSs with SRD of 1 and 2. The slack that resulted in the second case was exploited for energy reduction. We have used the algorithm pub-

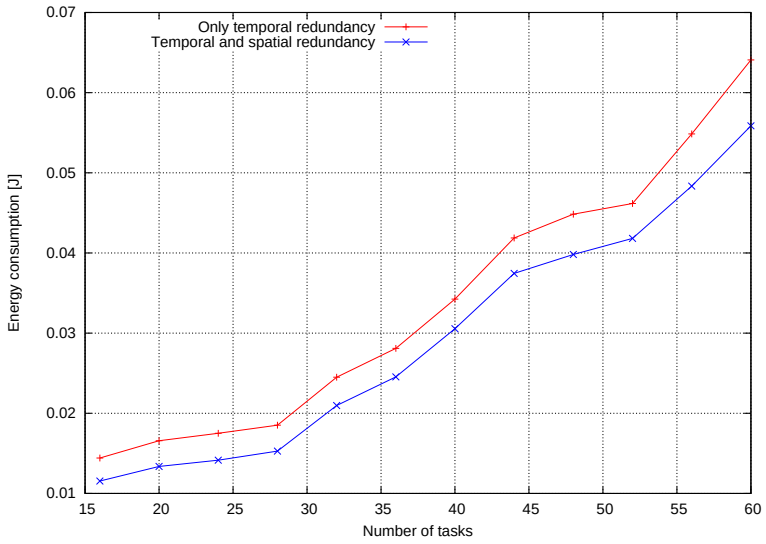


Figure 9.10: Energy consumption vs. number of tasks

lished in [ASE+04] for calculating the voltage levels for which to run the application. For our energy models, we considered a 70nm CMOS fabrication process. The resulted energy consumption is depicted in Figure 9.10. The energy reduction ranges from 20% to 13%. For this experiment, we considered the conservative scenario in which, at design time, an optimal voltage and/or frequency is computed for the whole NoC (see Eq. (9.1) in Section 9.5). We do not assume the availability of a dynamic voltage scaling capability in the NoC. If such capability existed, even larger energy savings could be achieved.

9.6.6 Real-Life Example: An Audio/Video Encoder

Finally, we applied our approach to a multimedia application, namely an audio/video encoder implementing the H.263 recommendation [Int05] of the International Telecommunication Union (ITU) for video encoding and the MPEG-1 Audio Layer 3 standard for audio encoding (ISO/IEC 11172-3 Layer 3 [Int93]).

Figure 9.11 depicts the task graph that models the application, while Figure 9.12 shows the application mapping to the NoC cores. The task partitioning, mapping, and profiling was done by Hu and Mărculescu [HM04b]. The video encoding part of the application consists of 9 tasks: frame prediction (FP), motion estimation (ME), discrete cosine transform (DCT), quantisation (Q), inverse quantisation (IQ), inverse discrete cosine transform (IDCT), motion compensation (MC), addition (ADD), and variable length encoding (VLE). Three memory regions are used for frame stores FS0, FS1, and FS2. The audio encoding part consists of 7 tasks: frame prediction (FP), fast Fourier transform (FFT), psycho-acoustic model (PAM), filter (Flt), modified discrete cosine transform (MDCT), and two iterative encoding tasks (IE1 and IE2). The numbers that annotate arcs in Figure 9.11 denote the communication amount of the message represented by the corresponding arc. The period of the task graph depends on the imposed frame rate, which depends on the video clip. We use periods of 41.6ms,

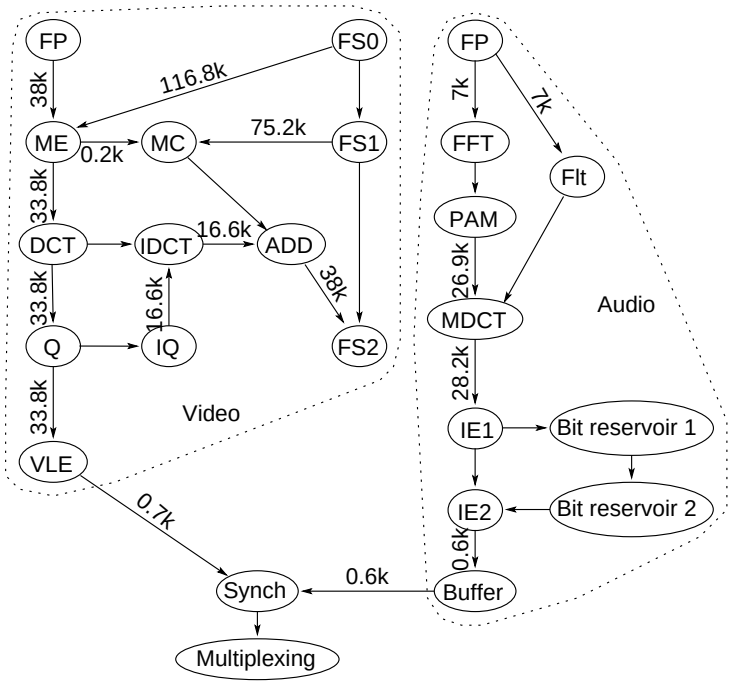


Figure 9.11: H.263 and MP3 encoding application

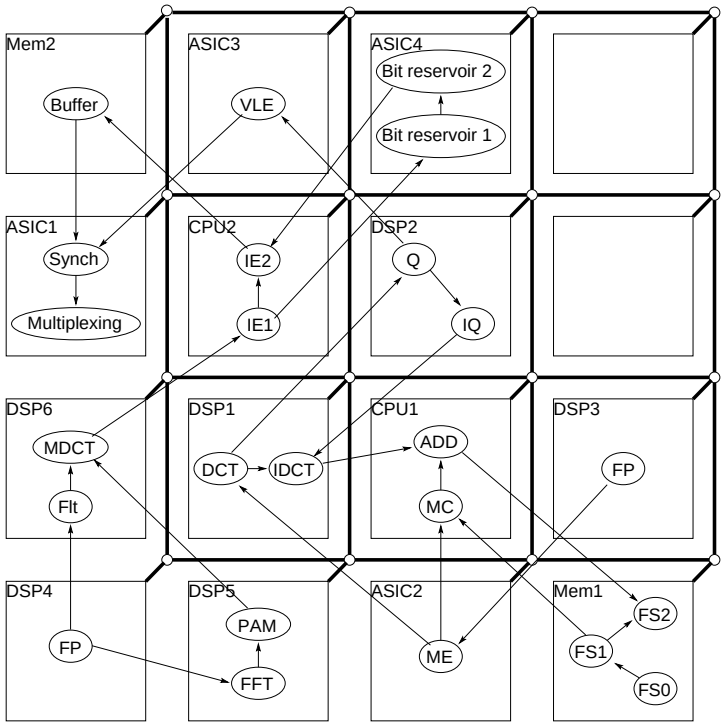


Figure 9.12: NoC implementation of the H.263 and MP3 encoding application

corresponding to 24 frames per second. The deadlines are equal to the periods.

The application is executed by an NoC with 6 DSPs, 2 CPUs, 4 ASICs, and 2 memory cores, organised as a 4×4 NoC with two unused tiles, as shown in Figure 9.12. The probability that a packet successfully traverses a network link is assumed to be 0.99. The approach combining spatially and temporally redundant message transmission obtained a 25% response time reduction relative to the approach deploying only temporal redundancy. The energy savings after voltage reduction amounted to 20%. Because of the relatively small design space of this example, the optimisation took only 3 seconds when combining spatially and temporally redundant communication supports.

9.7 Conclusions

In this chapter we addressed the problem of communication energy minimisation under task response time and message arrival probability constraints. The total communication energy is reduced by means of two strategies. On one hand, we intelligently select the communication supports of messages such that we reduce application response time with negligible energy penalty while satisfying message arrival probability constraints. On the other hand, the execution time slack can be exploited by deploying voltage and/or frequency scaling on the cores and communication links. The approach is efficient as it results in energy reductions up to 20%. Nevertheless, a significant cost and energy reduction potential has not been considered in this chapter, namely the reduction of buffer sizes at the network switches. The next chapter presents an approach for communication mapping with the goal to reduce the buffering need of packets while guaranteeing timeliness and lower bounds on message arrival probability.

Chapter 10

Buffer Space Aware Communication Synthesis for NoC

In this chapter we address two problems related to the buffering of packets at the switches of on-chip networks. First, we present an approach to minimise the buffer space demand of applications implemented on networks-on-chip. This is particularly relevant when designing application-specific NoCs, as the amount and distribution of on-chip memory can be tailored for the application. Second, we solve the problem of mapping the communication of an application implemented on an NoC with predefined buffers such that no buffer overflows occur during operation. Both problems are additionally constrained by timeliness requirements and bounds on the message arrival probability. For solving the described problems we introduce a buffer space demand analysis procedure, which we present in Section 10.3.3.

The buffer space demand minimisation is achieved by a combination of two techniques: an intelligent mapping of redundant messages to network links and a technique for delaying the sending of packets on links, also known as traffic shaping [RE02]. Section 10.1 gives a precise formulation of the two problems that we solve in this chapter. Section 10.2 discusses the two techniques that we propose. Section 10.3 presents our approach to solving the formulated problems and the buffer demand analysis procedure. Section 10.4 presents experimental results. Finally, Section 10.5 draws the conclusions.

10.1 Problem Formulation

In this section we define the two problems that we solve in this chapter.

10.1.1 Input

The input common to both problems consists of:

- The hardware model, i.e. the size of the NoC, and, for each link, the energy-per-bit, the bandwidth, and the probability of a packet to be successfully conveyed by the link;

- The application model, i.e. the set of task graphs Γ, the mapping of tasks to cores Map, the set of task periods Π_T, deadlines Δ_T, worst-case execution times, priorities and the amounts of data to be transmitted between communicating tasks;
- The communication model, i.e. the packet size and message priority for each message; and
- The lower bounds $B_{i,j}$ imposed on the message arrival probability $MAP_{i,j}$, for each message $\tau_i \rightarrow \tau_j$.

10.1.2 Constraints

The constraints for both problems are:

- All message arrival probabilities satisfy $MAP_{i,j} \geq B_{i,j}$;
- All tasks meet their deadlines.

10.1.3 Output

The communication synthesis problem with buffer space demand minimisation (CSBSDM) is formulated as follows:

Given the above input, for each message $\tau_i \rightarrow \tau_j$ find the communication support CS_{ij}, and determine the time each packet is delayed at each switch, such that the imposed constraints are satisfied and the total buffer space demand is minimised. Additionally, determine the needed buffer capacity of every input buffer at every switch.

The communication synthesis problem with predefined buffer space (CSPBS) is formulated as follows:

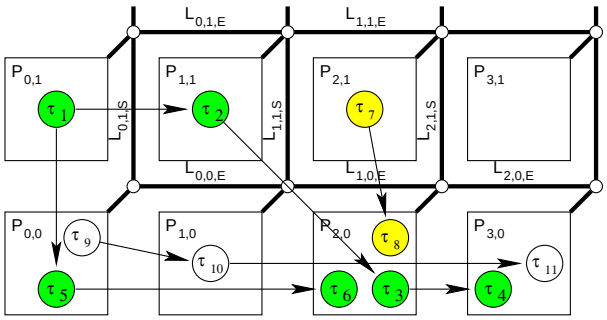
Given the above input, and additionally the capacity of every input buffer at every switch, for each message $\tau_i \rightarrow \tau_j$ find the communication support CS_{ij}, and determine the time each packet is delayed at each switch, such that the imposed constraints are satisfied and no buffer overflow occurs at any switch.

10.2 Motivational Example

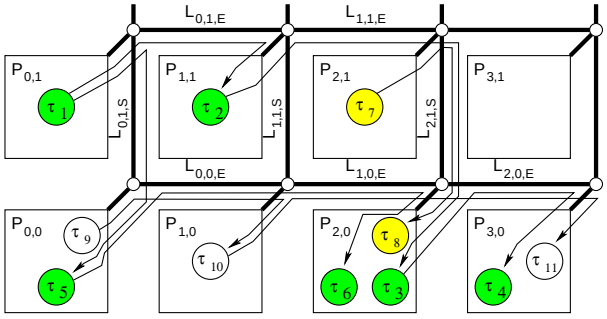
In this section we motivate the importance of intelligently choosing communication supports and we demonstrate the power of traffic shaping based on an example.

Let us consider the application shown in Figure 10.1(a). We assume that each message consists of a single packet. Assuming that messages are mapped on only shortest paths (paths traversing a minimum number of switches), for each message, except the message $\tau_2 \rightarrow \tau_3$, there is only one mapping alternative, namely the shortest path. For the message $\tau_2 \rightarrow \tau_3$, however, there are two such shortest paths, namely $L_{1,1,E} \rightarrow L_{2,1,S}$ and $L_{1,1,S} \rightarrow L_{1,0,E}$.

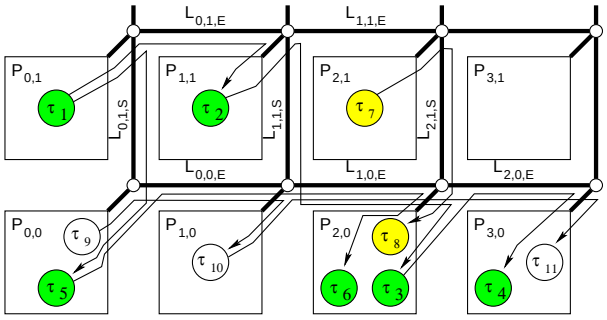
One way to minimise buffer space demand is to intelligently map the message $\tau_2 \rightarrow \tau_3$. Let us assume that the message is mapped on path $L_{1,1,E} \rightarrow L_{2,1,S}$. Such a situation is depicted in Figure 10.1(b). The corresponding Gantt diagram is shown in Figure 10.2(a). The rectangles represent task executions (respectively message transmissions) on the processing elements (respectively communication links) to which the tasks (messages) are mapped.



(a)

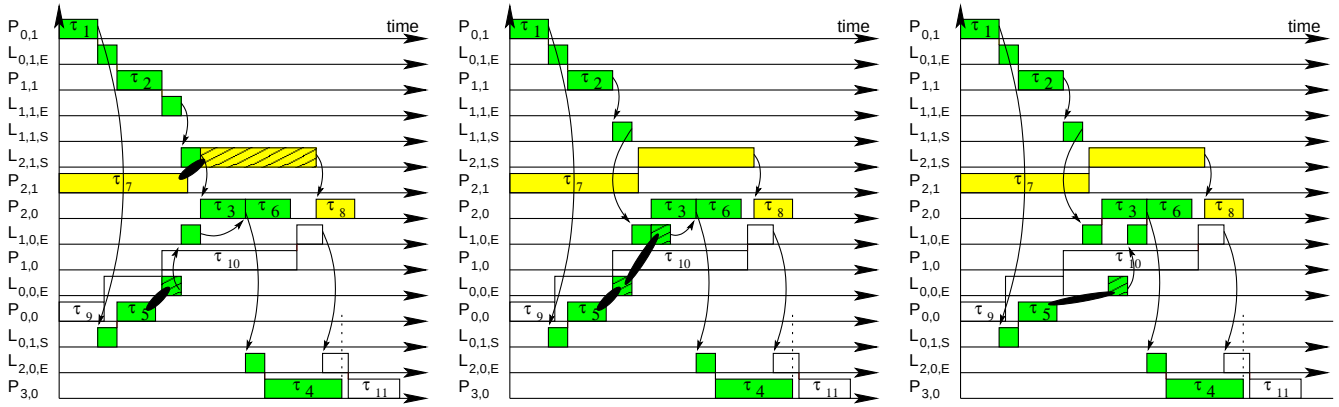


(b)



(c)

Figure 10.1: Application example



(a) (b) (c)

Figure 10.2: Impact of communication mapping and traffic shaping

Message $\tau_2 \rightarrow \tau_3$ competes with message $\tau_7 \rightarrow \tau_8$ for link $L_{2,1,S}$. Message $\tau_7 \rightarrow \tau_8$ arrives at the switch connecting tile $P_{2,1}$ to the network while message $\tau_2 \rightarrow \tau_3$ is conveyed on link $L_{2,1,S}$. Due to the unavailability of the link, message $\tau_7 \rightarrow \tau_8$ has to be buffered. The situations in which buffering is necessary are highlighted by black ellipses. Messages that have been buffered before being transmitted, due to momentary resource unavailability, are depicted in hashed manner. The total needed buffering space is proportional to the sum of hashed areas. One more such situation occurs in Figure 10.2(a), caused by the conflict between messages $\tau_5 \rightarrow \tau_6$ and $\tau_9 \rightarrow \tau_{10}$ on link $L_{0,0,E}$.

We observe that message $\tau_7 \rightarrow \tau_8$ needs a relatively large buffering space, which can be avoided by choosing a different mapping alternative for message $\tau_2 \rightarrow \tau_3$. This mapping is depicted in Figure 10.1(c), while its corresponding Gantt diagram is shown in Figure 10.2(b). However, while saving the buffering space required by message $\tau_7 \rightarrow \tau_8$, the new mapping introduces a conflict between messages $\tau_2 \rightarrow \tau_3$ and $\tau_5 \rightarrow \tau_6$ on link $L_{1,0,E}$. As a result, the packet from task τ_5 to task τ_6 has to be buffered at the switch S_{10} in the input buffer corresponding to link $L_{0,0,E}$. Nevertheless, because message $\tau_7 \rightarrow \tau_8$ does not need to be buffered, we reduced the overall buffer space demand relative to the alternative in Figure 10.1(b).

As there are no other mapping alternatives, we resort to the second technique, namely traffic shaping, in order to further reduce the total amount of buffering space.

In Figure 10.2(b), we observe that message $\tau_5 \rightarrow \tau_6$ is buffered twice, the first time before being sent on $L_{0,0,E}$, and the second time before being sent on link $L_{1,0,E}$. If we delayed the sending of message $\tau_5 \rightarrow \tau_6$, as shown in the Gantt diagram in Figure 10.2(c), we could avoid the need to buffer the message at switch S_{10}. In the particular case of our example, this message delaying comes with no task graph response time penalty. This is because the task graph response time is given by the largest response time among the tasks of the graph (τ_4 in our case), shown as the dotted line in Figure 10.2, which is unaffected by the delaying of message $\tau_5 \rightarrow \tau_6$. In general, traffic shaping may increase the application latency. Therefore, we deploy traffic shaping with predilection to messages on non-critical computation paths.

The above example demonstrates the efficiency of intelligent communication mapping and traffic shaping when applied to the problem of buffer need minimisation. Obviously, the techniques are also effective in the case of the second problem formulated in Section 10.1, the communication synthesis problem with predefined buffer space.

10.3 Approach Outline

The solution to both problems defined in Section 10.1 consists of two components each: the set of message communication supports and the set of packet delays. Thus, each problem is divided into two subproblems, the communication mapping subproblem (CM), which determines the communication support for each message, and the traffic shaping subproblem (TS), which determines the possible delays applied to forwarding a particular packet. Depending on the actual problem, we will introduce CSBSDM-CM and CSBSDM-TS, and CSPBS-CM and CSPBS-TS respectively.

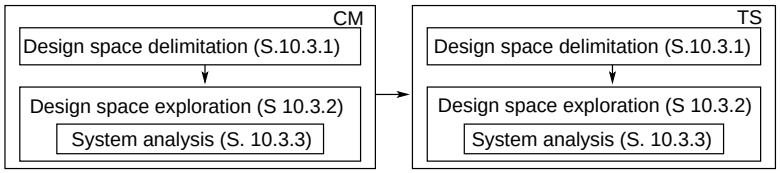


Figure 10.3: Approach outline

The outline of our approach is depicted in Figure 10.3. Solving the communication mapping as well as the traffic shaping subproblem is itself decomposed into three subproblems:

1. Delimit the space of potential solutions (Section 10.3.1)
2. Deploy an efficient strategy for the exploration of the design space (Section 10.3.2), and
3. Find a fast and accurate system analysis procedure for guiding the search (Section 10.3.3).

10.3.1 Delimitation of the Design Space

Concerning the CM problem, including all possible CSs for each message in the set of potential solutions leads to a very large design space, impossible to explore in reasonable time. Thus, in Section 9.3 we established criteria for picking only promising CS candidates, which we include in the space of potential solutions.

The solution space for the TS problem is constructed as follows. For each tuple $(p_{i,j}, S)$, where $p_{i,j}$ is a packet from task τ_i to task τ_j and S is a network switch on its route, we consider the set of delays $\{0, \Delta, 2\Delta, \ldots, D_j\}$, where Δ is the minimum amount of time it takes for the packet to traverse a network link, and $D_j = \delta_j - WCET_j - H \cdot \Delta$, where δ_j is the deadline of task τ_j, $WCET_j$ is the worst-case execution time of task τ_j, and H is the Manhattan distance between the two cores on which tasks τ_i and τ_j are mapped. Delaying the packet $p_{i,j}$ longer than D_j would certainly cause task τ_j to break its deadline δ_j if it executed for its worst-case execution time $WCET_j$.

10.3.2 Exploration Strategy

Cost Function

The value of the cost function that drives the design space exploration is infinite for solutions in which there exists a task whose response time exceeds its deadline.

The cost function for the CSBSDM-CM and CSBSDM-TS subproblems is $\sum_{b \in B} d_b$, where B is the set of all switch input buffers, b is a buffer in this set, and d_b is the maximum demand of buffer space of the application at buffer b.

The cost function for the CSPBS-CM and CSPBS-TS subproblems is $max_{b \in B}(d_b - c_b)$, where c_b is the capacity of buffer b. Solutions of the CSPBS problem with strictly positive cost function value do not satisfy the buffer space constraint and are thus unfeasible. For the CSPBS problem, we stop the design space exploration as soon as we find a solution whose cost is zero or negative.

```
(1)  sm = sort_messages;
(2)  for each msg in sm do
(3)    CS[msg] = select(msg, candidates[msg]);
(4)    if CS[msg] =NONE then
(5)      abort NO SOLUTION;
(6)  return CS;

     select(msg, cand_list):
(7)  cost = ∞; selected =NONE;
(8)  for each cnd in cand_list do
(9)    CS[msg] = cnd; crt_cost = cost_func;
(10)   if crt_cost < cost then
(11)     selected = cnd; cost = crt_cost;
(12) return selected;
```

Figure 10.4: Heuristic for communication mapping

Communication Mapping

We propose a greedy heuristic for communication mapping. We map messages to CSs stepwise. At each step, we map one message and we obtain a partial solution. When evaluating partial solutions, the messages that have not yet been mapped are not considered.

The heuristic proceeds as shown in Figure 10.4, lines 1–6. It returns the list of communication supports for each message if a feasible solution is found (line 6) or aborts otherwise (line 5). Before proceeding, we sort all messages in increasing order of their number of mapping alternatives (line 1). Then, we iterate through the sorted list of messages sm. In each iteration, we select a mapping alternative to the current message (line 3).

The selection of a mapping alternative out of the list of candidates (determined in the previous step, Section 10.3.1, and in Section 9.3) is shown in Figure 10.4, lines 7–12. We iterate over the list of mapping alternatives (line 8) and evaluate each of them (line 9). We select the alternative that gives the minimum cost (line 11).

The motivation for synthesizing the communication in the particular order of increasing number of mapping alternatives of messages is the following. We would like to minimise the chance that the heuristic runs into the situation in which it does not find any feasible solution, although at least one exists. If messages enjoying a large number of mapping alternatives are mapped first, we restrict the search space prematurely and gratuitously, running the risk that no feasible mapping is found for other messages among their few mapping alternatives.

Traffic Shaping

The greedy heuristic, shown in Figure 10.5, determines the amount of time each communication task has to be delayed (a.k.a. shaping delay). As a first step, we sort the communication tasks according to a criterion to be explained later (line 1). Then, for all communication tasks in the sorted list we find the appropriate shaping delay (line 2). The selection of a shaping delay of a communication task is performed by the function $shape$ (lines 3–9). We probe shaping delays ranging from 0 to $D_j = \delta_j - WCET_j - H \cdot \Delta$ in increments of Δ, where the index δ_j and $WCET_j$

```
(1)  sct =sort_comm_tasks;
(2)  for each τ in sct do delay[τ] =shape(τ);
     shape(τ):
(3)  cost = ∞;
(4)  for delay[τ] = 0; delay[τ] < D_τ; delay[τ] := delay[τ] + Δ
(5)    crt_cost = cost_func;
(6)    if crt_cost < cost then
(7)      best_delay = delay[τ]; cost = crt_cost;
(8)  end for;
(9)  return best_delay;
```

Figure 10.5: Heuristic for traffic shaping

are the deadline and the worst-case execution time of the receiving task τ_j (see Section 10.3.1). For each probed shaping delay, we evaluate the cost of the obtained partial solution (line 5). When calculating it, we assume that the shaping delay of those tasks for which none has yet been chosen is 0. We select the shaping delay that leads to the minimum cost solution (lines 6–7).

Before closing this section, we will explain in which order to perform the shaping delay selection. We observe that communication tasks on paths whose response times are closer to the deadline have a smaller potential for delaying. Thus, delaying such communication tasks runs a higher risk to break the timeliness constraints. In order to quantify this risk, we compute the worst-case response time R_τ of each leaf task τ. Then, for each task τ_i we determine $\mathcal{L}(\tau_i)$, the set of leaf tasks τ_j such that there exists a computation path between task τ_i and τ_j. Then, to each task τ_i we assign the value $prt_i = min_{\tau \in \mathcal{L}(\tau_i)}(\delta_\tau - R_\tau)$. Last, we sort the tasks in decreasing order of their prt_i.[1] In case of ties, tasks with smaller depths[2] in the task graph are placed after tasks deeper in the graph. (If tasks with small depths were delayed first, their delay would seriously restrict the range of feasible delays of tasks with large depths.)

10.3.3 System Analysis Procedure

In order to be able to compute the cost function as defined in Section 10.3.2, we need to determine the worst-case response time of each task as well as the buffering demand at each buffer in the worst case. To do so, we extended the schedulability analysis algorithm of Palencia and González [PG98].

At the core of the worst-case response time calculation of task τ_i is a fix-point equation of type $w_i = R_i(w_i)$. $R_i(t)$ gives the worst-case response time of task τ_i when considering interference of tasks of higher priority than that of τ_i that arrive in the interval $[0, t)$. The time origin is considered the arrival time of task τ_i. Thus, evaluating R_i at two time moments, t_1 and t_2, allows us to determine the execution time demanded by higher priority tasks arrived during the interval $[t_1, t_2)$. More details regarding the calculation of the worst-case response time

[1] The procedure can be easily generalised for the case in which not only leaf tasks have deadlines.

[2] The depth of a task τ is the length of the longest computation path from a root task to task τ.

(1) $Buf = 0; b = 0; t = 0; F = R_0(t); F_1 = F;$
(2) **loop**
(3) $t' = next\_t';$
(4) $F' = R_0(t');$
(5) **if** $t' = F'$ **then**
(6) **return** $Buf;$
(7) $b' = (F' - F) \cdot bw + b - (t < F_1)?1 : 0;$
(8) **if** $b' > Buf$ **then**
(9) $Buf = b';$
(10) **if** $t' > F_1$ **then**
(11) $b := b - (t' - max(t, F_1) - (F' - F)) \cdot bw;$
(12) $t = t'; F = F';$
(13) **end loop**;

Figure 10.6: Buffer space analysis algorithm

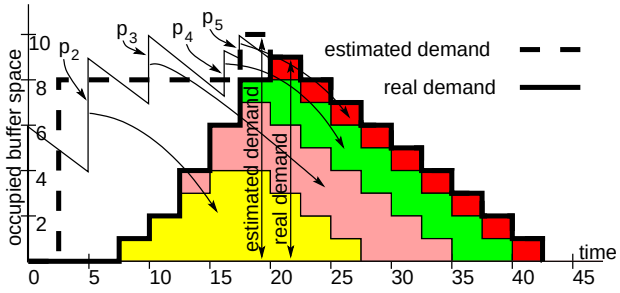


Figure 10.7: Waiting time and buffer demand

can be found in cited work [PG98]. Here we will concentrate on our approach to buffer demand analysis. For communication tasks, their "execution time" on their "processors" are actually the transmission times of packets on network links. This transmission time is proportional to the length of the packet. Thus, by means of the analysis of Palencia and González, which can determine the execution time demanded during a time interval, we are able to determine the buffering demand arrived during the interval.

The algorithm for the calculation of the buffer space demand of an ingress buffer of an arbitrary network link is given in Figure 10.6. We explain the algorithm based on the following example.

Let us consider the following scenario. Prior to time moment 0, a 400MHz link is idle. The links convey the bits of a word in parallel, with one word per cycle. At time moment 0, the first word of a 6-word packet p_1 arrives at the switch and is immediately conveyed on the link without buffering. The following packets subsequently arrive at the switch and demand forwarding on the link: p_2, 5 words long, arrives at 5ns, p_3, 3 words long, arrives at 10ns, p_4, 2 words long, arrives at 15.25ns, and p_5, 1 word long, arrives at 17.5ns. Let us assume that a fictive packet p_0 of zero length and of very low priority arrived at time 0^+, i.e. immediately after time 0. We compute the worst-case buffer space need based on the worst-case transmission time of this fictive packet.

The scenario is shown in Figure 10.7. Time is shown on the abscissa, while the saw-teeth function shows the instantaneous commu-

nication time backlog, and the solid step function shows the instantaneous amount of occupied buffer space. The arrows pointing from the steps in the backlog line to the shaded areas show which message arrival causes the corresponding buffering.

The time interval during which the link is busy sending packets is called the busy period. In our example the busy period is the interval $[0, 42.5)$, as can be seen on the figure. The main part of the algorithm in Figure 10.6 consists of a loop (lines 2–13). A subinterval $[t, t')$ of the busy period is considered in each iteration of the loop. In the first iteration $t = 0$ while in iteration i, t takes the value of t' of iteration $i - 1$ for all $i > 1$ (line 12). F and F' are the times at which the link would be idle if it has to convey just the packets arrived sooner than or exactly at times t and t' respectively (lines 1 and 4). t', the upper limit of the interval under consideration in each iteration, is obtained as shown in line 3. For the moment, let us consider that $next\_t' = F$ and we will discuss the rationale and other possible choices later in the section.

For our example, only packet p_1 of 6 words is to be sent just after time 0. Hence, $R_0(0^+) = 6\text{words}/0.4 \cdot 10^{-9}\text{words/sec} = 15\text{ns}$. The first iteration of the loop considers the interval $[t = 0, t' = F = R_0(0^+) = 15)$. We compute $F' = R_0(t' = 15)$ (line 4) and we get 35ns, i.e. the 15ns needed to convey the six words of packet p_1 plus the $5\text{words}/0.4 \cdot 10^{-9}\text{words/sec} = 12.5\text{ns}$ needed to convey packet p_2 plus the 7.5ns needed to convey packet p_3 (p_2 and p_3 having arrived in the interval $[0, 15)$). The time by which the link would become idle if it has to convey just the packets arrived prior to $t' = 15\text{ns}$ is greater than t'. Hence, there are unexplored parts of the busy period left and the buffer space calculation is not yet over (lines 5–6). The packets that arrived between 0 and 15ns extended the busy period with $F' - F = 20\text{ns}$, hence the number of newly arrived words is $(F' - F) \times bw = 20\text{ns} \times 0.4 \cdot 10^{-9} = 8\text{words}$. The algorithm is unable to determine the exact time moments when the 8 words arrived. Therefore, we assume the worst possible moment from the perspective of the buffer space demand. This moment is time t^+, i.e. immediately after time t. The 8 words are latched at the next clock period after time $t^+ = 0$, i.e. at 2.5ns. b', the amount of occupied buffer after latching, is b, the amount of occupied buffer at time t, plus the 8 words, minus possibly one word that could have been pumped out of the buffer between t and $t + 2.5\text{ns}$. During the time interval $[0, F_1 = 15)$, where F_1 is the time it takes to convey packet p_1, the words conveyed on the link belong to p_1, which is not stored. Therefore, no parts of the buffer are freed in the interval $[0, F_1)$ (see line 7). If the required buffer space is larger than what has been computed so far, the buffer space demand is updated (lines 8–9). Because no buffer space is freed during the interval $[0, 15)$, lines 10–11 are not executed in the first iteration of the loop.

The second iteration considers the interval $[t = 15, t' = 35)$. $F = 35\text{ns}$ and $F' = 42.5\text{ns}$ in this case. Hence, $(F' - F) \cdot bw = 7.5\text{ns} \times 0.4 \cdot 10^{-9}\text{words/sec} = 3\text{words}$ arrived during interval $[15, 35)$. The three words are considered to have arrived at the worst moment, i.e. at 15^+. They are latched at time 17.5ns when $b = 8 - 1$, i.e. the 8 words that are stored in the buffer at 15ns minus one word that is pumped out between 15 and 17.5ns. Thus b', the amount of occupied buffer at 17.5ns is $8 - 1 + 3 = 10$ (line 7). The value Buf is updated accordingly (lines

8–9). Between 15 and 35ns some words that were stored in the buffer are sent on the link and therefore we have to account for the reduction of the amount of occupied buffer. Thus, the amount of occupied buffer at 35ns is equal to 8, the amount present at 15ns, plus the 3 words that arrived between 15 and 35ns and minus the $(35 - 15) \times 0.4 \cdot 10^{-9} = 8$ that are conveyed on the link in the interval $[15, 35)$ (see lines 10–11).

The third iteration considers the interval $[35, 42.5)$. As no new packets arrive during this interval, $t' = R_0(t') = 42.5$ and the algorithm has reached a fix-point and returns the value of Buf.

We will close the section with a discussion on $next\_t'$, the complexity of the algorithm, and the trade-off between the algorithm execution speed and accuracy.

The actual amount of occupied buffer is shown as the thick solid line in Figure 10.7, while the amount, as estimated by the algorithm, is shown as thick dotted line. We observe that the analysis procedure produces a pessimistic result. This is due to the fact that the analysis assumes that the new packets that arrive in the interval $[t, t')$ arrive always at the worst possible moment, that is moment t^+. If we partitioned the interval in which the link is busy sending packets into many shorter intervals, we could reduce the pessimism of the analysis, because fewer arrivals would be amassed at the same time moment. However, that would also imply that we invoke function R_0 more often, which is computationally expensive. Thus, there exists a trade-off between speed of the analysis and pessimism, which is reflected in the choice of $next\_t'$ (line 3). A value closer to t would lead to short intervals, i.e. less pessimism and slower analysis, while a value farther from t would lead to longer intervals, i.e. more pessimistic but possibly (not necessarily, as shown below) faster analysis.

In our experiments, we use $next\_t' = F$, which is the finishing time of the busy period if no new packets arrive after time t. Choosing a value larger than F would incur the risk to overestimate the busy period. As a result, packets that arrive after the real finishing time of the busy period might wrongly be considered as part of the current busy period. On one hand that leads to the overestimation of the buffer space, and on the other hand it increases the time until the loop in Figure 10.6 reaches fix-point. In our experiments, choosing $next\_t' = 1.6 \cdot F$ results in a 10.3% buffer overestimation and a $2.3\times$ larger analysis time relative to the case when $next\_t' = F$. Conversely, choosing smaller values for $next\_t'$ lead to reductions of at most 5.3% of the buffer space estimate while the analysis time increased with up to 78.5%.

The algorithm is of pseudo-polynomial complexity due to the calculation of function R [PG98].

10.4 Experimental Results

We use a set of 225 synthetic applications in order to assess the efficiency of our approach to solve the CSBSDM problem. The applications consist of 27 to 79 tasks, which are mapped on a 4×4 NoC. The probability that a 110-bit packet traverses one network link unscrambled is 0.99, while the imposed lower bound on the message arrival probability is also 0.99. Due to the fact that the implementation of the packet delay capability could excessively increase the complexity of the switches,

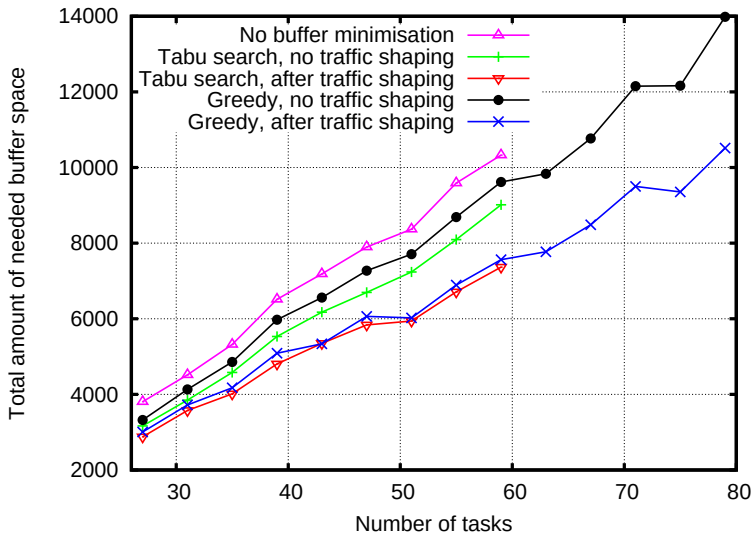


Figure 10.8: Buffering space vs. number of tasks

we have considered that traffic shaping is performed only at the source cores. This has the advantage of no hardware overhead.

10.4.1 Evaluation of the Solution to the CSBSDM Problem

For each application, we synthesized the communication using three approaches and we determined the total buffer space demand obtained in each of the three cases. In the first case, we use the buffer space minimisation approach presented in chapter. In the second case, we replaced the greedy heuristics described in Section 10.3.2 with Tabu Search based heuristics that are assumed to generate close to optimal solutions provided that they are let to explore the design space for a very long time. In the third case, we deployed the communication synthesis approach presented in the previous chapter, in which we do not considered buffer space minimisation. The resulting total buffer space as a function of the number of tasks is shown in Figure 10.8 as the curves labelled with “greedy”, “tabu”, and “no buffer minimisation” respectively.

First, we observe that buffer space minimisation is worth pursuing, as it results in 22.3% reduction of buffer space on average when compared to the case when buffer space minimisation is neglected. Second, traffic shaping is an effective technique, reducing the buffer space demand with 14.2% on average relative to the approach that is based solely on communication mapping. Third, the greedy heuristic performs well as it obtains results on average of only 3.6% worse than the close-to-optimal Tabu Search. The running times of the Tabu Search based and the greedy heuristic, as measured on a 1533 MHz AMD Athlon processor, are shown in Figure 10.9. The greedy heuristic performs about two orders of magnitude faster (note the logarithmic scale of the y axis) than the Tabu Search based heuristic. Thus, we are able to synthesize the communication for applications of 79 tasks

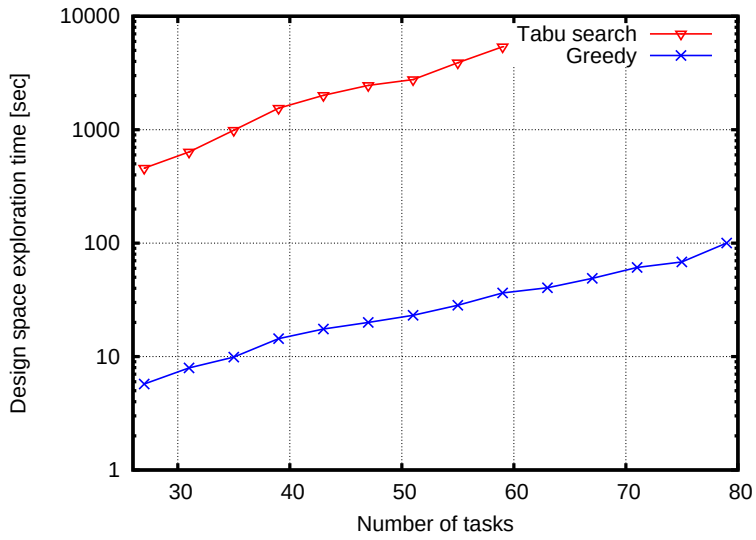


Figure 10.9: Run time comparison

in $1'40''$, while the Tabu Search based heuristic requires around $1h30'$ for applications of 59 tasks.

10.4.2 Evaluation of the Solution to the CSPBS Problem

We use 50 different 4×4 NoCs in order to assess the efficiency of our approach to solve the CSPBS problem. The total buffering capacities at switches range between $9,000$ and $30,000$ bits, uniformly distributed among the switches. We map 200 applications, one at a time, each consisting of 40 tasks, on each of the 50 NoCs, and we attempt to synthesize the communication of the application such that no buffer overflows or deadline violations occur. For each NoC, we count the applications for which we succeeded to find feasible solutions to the CSPBS problem. The percentage of the number of applications for which feasible communication synthesis solutions were found is plotted as a function of the total buffer capacity of the NoC in Figure 10.10. The proposed heuristic soundly outperforms the approach that neglects the buffering aspect as the percentage of found solutions is on average 53 points higher in the former case than in the latter. Also, the deployment of traffic shaping results in leveraging the percentage of found solutions to the CSPBS problem with 18.5% compared to the case when no traffic shaping is deployed. The results of the greedy heuristic come within 9% of the results obtained by Tabu Search, while the greedy heuristic runs on average 25 times faster.

10.4.3 Real-Life Example: An Audio/Video Encoder

Finally, we applied our approach the multimedia application described in Section 9.6.6 and depicted in Figures 9.11 and 9.12. The communication mapping heuristic reduced the total buffer space with 12.6% relative to the approach that synthesized the communication without attempting to reduce the total buffer space demand. Traffic shaping

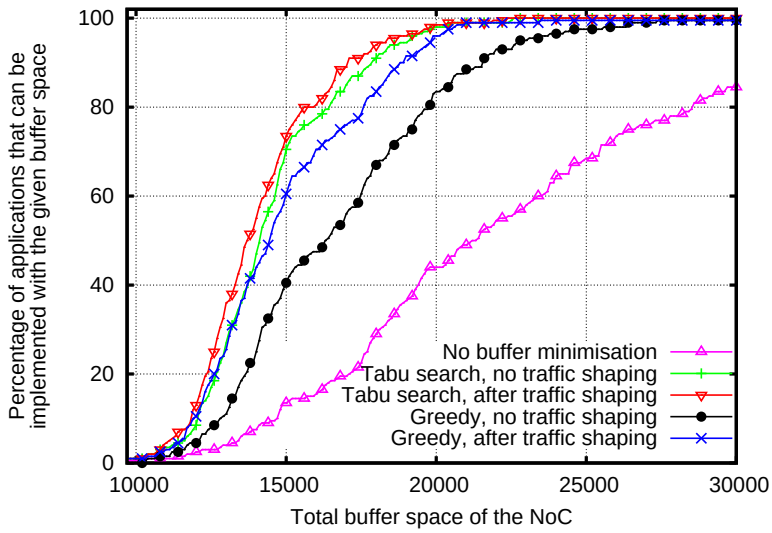


Figure 10.10: Percentage of the number of feasible applications as a function of the NoC buffer capacity

allowed for a further reduction of 31.8%, giving a total buffer space demand of 77.3kB.

10.5 Conclusions

In this chapter, we developed an approach to the worst-case buffer need analysis of time constrained applications implemented on NoCs. Based on this analysis we solved two related problems: (1) the total buffer space need minimisation for application-specific NoCs and (2) communication synthesis with imposed buffer space constraints. For both cases we guarantee that imposed deadlines and message arrival probability thresholds are satisfied. We have argued that traffic shaping is a powerful method for buffer space minimisation. We proposed two efficient greedy heuristics for the communication mapping and traffic shaping subproblems and we presented experimental results, which demonstrate the efficiency of the approach.

Part IV

Conclusions

Chapter 11

Conclusions

This thesis addresses several problems related to real-time systems with stochastic behaviour. Two sources of stochastic behaviour were considered. The first source, namely the stochastic task execution times, stems with predilection from the application, although features of the hardware platform, such as cache replacement algorithms, may also influence it. The second source, namely the transient faults that may occur on the on-chip network links, is inherent in the hardware platform and the environment.

11.1 Applications with Stochastic Execution Times

In the area of real-time systems with stochastic task execution times, we provide three different analysis approaches, each efficiently applicable in a different context. Additionally, we propose a heuristic for deadline miss probability minimisation.

11.1.1 An Exact Approach for Deadline Miss Ratio Analysis

In Chapter 4 we proposed a method for the schedulability analysis of task sets with probabilistically distributed task execution times. Our method improves the currently existing ones by providing *exact* solutions for *larger* and *less restricted* task sets. Specifically, we allow arbitrary continuous task execution time probability distributions, and we do not restrict our approach to one particular scheduling policy. Additionally, task dependencies are supported, as well as arbitrary deadlines.

The analysis of task sets under such generous assumptions is made possible by three complexity management methods:

1. The exploitation of the PMI concept,
2. The concurrent construction and analysis of the stochastic process, and
3. The usage of a sliding window of states, made possible by the construction in topological order.

As the presented experiments demonstrate, the proposed method can efficiently be applied to applications implemented on monoprocessor systems.

11.1.2 An Approximate Approach for Deadline Miss Ratio Analysis

In Chapter 5 we presented an approach to performance analysis of tasks with probabilistically distributed execution times, implemented on multiprocessor systems. The arbitrary probability distributions of the execution times are approximated with Coxian distributions, and the expanded underlying Markov chain is constructed in a memory efficient manner exploiting the structural regularities of the chain. In this way we have practically pushed the solution of an extremely complex problem to its limits. Our approach also allows to trade-off between time and memory complexity on one side and solution accuracy on the other. The efficiency of the approach has been investigated by means of experiments. The factors that influence the analysis complexity, and their quantitative impact on the analysis resource demands have been discussed. Additional extensions of the problem formulation and their impact on complexity have also been illustrated.

11.1.3 Minimisation of Deadline Miss Ratios

In Chapter 6 we addressed the problem of design optimisation of soft real-time systems with stochastic task execution times under deadline miss ratio constraints. The contribution is threefold:

1. We have shown that methods considering fixed execution time models are unsuited for this problem.
2. We presented a design space exploration strategy based on tabu search for task mapping and priority assignment.
3. We introduced a fast and approximate analysis for guiding the design space exploration.

Experiments demonstrated the efficiency of the proposed approach.

11.2 Transient Faults of Network-on-Chip Links

The contribution of this thesis in the area of network-on-chip communication in the presence of transient faults on the links is fourfold. First, we present a way to intelligently combine spatial and temporal redundant communication for response time reduction and energy minimisation. Second, we provide an analysis algorithm that determines the amount of needed buffers at the network switches. Third, we present a heuristic algorithm for minimising the buffer space demand of applications. Fourth, we propose a heuristic algorithm for communication mapping under buffer space constraints.

11.2.1 Time-Constrained Energy-Efficient Communication Synthesis

In Chapter 9 we presented an approach to reliable, low-energy on-chip communication for time-constrained applications implemented on NoC. The contribution is manifold:

1. We showed how to generate supports for message communication in order to meet the message arrival probability constraint and to minimise communication energy.
2. We gave a heuristic for selecting most promising communication supports with respect to application responsiveness and energy.
3. We modelled the fault-tolerant application for response time analysis.
4. We presented experiments demonstrating the proposed approach.

11.2.2 Communication Buffer Minimisation

In Chapter 10 we developed an approach to the worst-case buffer need analysis of time-constrained applications implemented on NoCs. Based on this analysis we solved two related problems:

1. The total buffer space need minimisation for application-specific NoCs, and
2. the communication synthesis with imposed buffer space constraints.

For both cases we guarantee that imposed deadlines and message arrival probability thresholds are satisfied. We argued that traffic shaping is a powerful method for buffer space minimisation. We proposed two efficient greedy heuristics for the communication mapping and traffic shaping subproblems and we presented experimental results, which demonstrate the efficiency of the approach.

Appendix A

Abbreviations

AA	Approximate Analysis
BCET	Best-Case Execution Time
CGPN	Concurrent Generalised Petri Nets
CM	Communication Mapping
CS	Communication Support
CSBSDM	Communication Synthesis with Buffer Space Demand Minimisation
CSPBS	Communication Synthesis with Predefined Buffer Space
CTMC	Continuous Time Markov Chain
ECE	Expected Communication Energy
ENS	Exhaustive Neighbourhood Search
ETPDF	Execution Time Probability Density Function
GRD	General Redundancy Degree
GSMP	Generalised Semi-Markov Process
GSPN	Generalised Stochastic Petri Net
LCM	Least Common Multiple
LO-AET	Laxity Optimisation based on Average Execution Times
MAP	Message Arrival Probability
MRGP	Markov Regenerative Process
MRSPN	Markov Regenerative Stochastic Petri Net
PA	High-Complexity Performance Analysis
PMI	Priority Monotonicity Interval
RNS	Restricted Neighbourhood Search
SRD	Spatial Redundancy Degree
TRD	Temporal Redundancy Degree
TRG	Tangible Reachability Graph
TS	Traffic Shaping
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Bibliography

[AB98] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium*, pages 123–132, 1998.

[AB99] L. Abeni and G. Butazzo. QoS guarantee using probabilistic deadlines. In *Proceedings of the* 11^{th} *Euromicro Conference on Real-Time Systems*, pages 242–249, 1999.

[ABD+91] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the* 8^{th} *IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[ABD+95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Journal of Real-Time Systems*, 8(2-3):173–198, March-May 1995.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[ABRW93] N. C Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. *Computer Systems Science and Engineering*, 8(3):80–89, 1993.

[AKK+00] K. Aingaran, F. Klass, C. M. Kim, C. Amir, J. Mitra, E. You, J. Mohd, and S. K. Dong. Coupling noise analysis for VLSI and ULSI circuits. In *Proceedings of IEEE ISQED*, pages 485–489, 2000.

[ASE+04] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. Al-Hashimi. Simultaneous communication and processor voltage scaling for dynamic and leakage energy reduction in time-constrained systems. In *Proc. of ICCAD*, 2004.

[Aud91] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Department of Computer Science, University of York, December 1991.

[BBB01] E. Bini, G. Butazzo, and G. Butazzo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the* 13^{th}

Euromicro Conference on Real-Time Systems, pages 59–66, 2001.

[BBB03] A. Burns, G. Bernat, and I. Broster. A probabilistic framework for schedulability analysis. In R. Alur and I. Lee, editors, *Proceedings of the Third International Embedded Software Conference, EMSOFT*, number LNCS 2855 in Lecture Notes in Computer Science, pages 1–15, 2003.

[BBD02] D. Bertozzi, L. Benini, and G. De Micheli. Low power error resilient encoding for on-chip data buses. In *Proc. of DATE*, pages 102–109, 2002.

[BBRN02] I. Broster, A. Burns, and G. Rodriguez-Navas. Probabilistic analysis of CAN with faults. In *Proceedings of the* 23^{rd} *Real-Time Systems Symposium*, 2002.

[BCFR87] G. Balbo, G. Chiola, G. Franceschinis, and G. M. Roet. On the efficient construction of the tangible reachability graph of Generalized Stochastic Petri Nets. In *Proceedings of the* 2^{nd} *Workshop on Petri Nets and Performance Models*, pages 85–92, 1987.

[BCKD00] P. Buchholtz, G. Ciardo, P. Kemper, and S. Donatelli. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS Journal on Computing*, 13(3):203–222, 2000.

[BCP02] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the* 23^{rd} *Real-Time Systems Symposium*, pages 279–288, 2002.

[BD02] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[BIGA04] E. Bolotin, Cidon I., R. Ginosar, and Kolodny A. QNoC: QoS architecture and design process for networks-on-chip. *Journal of Systems Architecture*, 50:105–128, 2004.

[Bla76] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In E. Gelenbe and H. Bellner, editors, *Modeling and Performance Evaluation of Computer Systems*. North-Holland, Amsterdam, 1976.

[Bos91] Bosch, Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany. *CAN Specification*, 1991.

[BPSW99] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the* 7^{th} *International Working Conference on Dependable Computing for Critical Applications*, pages 339–356, 1999.

[But97] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic, 1997.

[BW94] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages*. Addison Wesley, 1994.

[CKT94] H. Choi, V.G. Kulkarni, and K.S. Trivedi. Markov Regenerative Stochastic Petri Nets. *Performance Evaluation*, 20(1–3):337–357, 1994.

[Cox55] D. R. Cox. A use of complex probabilities in the theory of stochastic processes. In *Proceedings of the Cambridge Philosophical Society*, pages 313–319, 1955.

[CXSP04] V. Chandra, A. Xu, H. Schmit, and L. Pileggi. An interconnect channel design methodology for high performance integrated circuits. In *Proceedings of the Conference on Design Automation and Test in Europe*, page 21138, 2004.

[Dal99] W. Dally. Interconnect-limited VLSI architecture. In *IEEE Conference on Interconnect Technologies*, pages 15–17, 1999.

[Dav81] M. Davio. Kronecker products and shuffle algebra. *IEEE Transactions on Computing*, C-30(2):1099–1109, 1981.

[DGK+02] J. L. Díaz, D. F. García, K. Kim, C.-G. Lee, L. Lo Bello, J. M. López, S. L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the* 23^{rd} *Real-Time Systems Symposium*, 2002.

[Die00] K. Diefenderhoff. Extreme lithography. *Microprocessor Report*, 6(19), 2000.

[dJG00] G. de Veciana, M. Jacome, and J.-H. Guo. Assessing probabilistic timing constraints on system performance. *Design Automation for Embedded Systems*, 5(1):61–81, February 2000.

[DLS01] B. Doytchinov, J. P. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with earliest-deadline-first queue discipline. *Annals of Applied Probability*, 11:332–378, 2001.

[DM03] T. Dumitraş and R. Mărculescu. On-chip stochastic communication. In *Proc. of DATE*, 2003.

[DRGR03] J. Dielissen, A. Rădulescu, K. Goossens, and E. Rijpkema. Concepts and implementation of the Philips network-on-chip. In *IP-Based SoC Design*, 2003.

[DW93] J. G. Dai and Y. Wang. Nonexistence of Brownian models for certain multiclass queueing networks. *Queueing Systems*, 13:41–46, 1993.

[Ele02] P. Eles. System design and methodology, 2002. http://www.ida.liu.se/~TDTS30/.

[Ern98] R. Ernst. Codesign of embedded systems: Status and trends. *IEEE Design and Test of Computers*, pages 45–54, April-June 1998.

[ETS] European telecommunications standards institute. http://www.etsi.org/.

[Fid98] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Journal of Real-Time Systems*, 14(1):61–93, 1998.

[FJ98] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[Gar99] M.K. Gardner. *Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[Gau98] H. Gautama. A probabilistic approach to the analysis of program execution time. Technical Report 1-68340-44(1998)06, Faculty of Information Technology and Systems, Delft University of Technology, 1998.

[GI99] A. Goel and P. Indyk. Stochastic load balancing and related problems. In *IEEE Symposium on Foundations of Computer Science*, pages 579–586, 1999.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[GKL91] M. González Harbour, M. H. Klein, and J. P. Lehoczky. Fixed priority scheduling of periodic rasks with varying execution priority. In *Proceedings of the IEEE Real Time Systems Symposium*, pages 116–128, 1991.

[GL94] R. German and C. Lindemann. Analysis of stochastic Petri Nets by the method of supplementary variables. *Performance Evaluation*, 20(1–3):317–335, 1994.

[GL99] M.K. Gardner and J. W.S. Liu. *Analyzing Stochastic Fixed-Priority Real-Time Systems*, pages 44–58. Springer, 1999.

[Glo89] F. Glover. Tabu search—Part I. *ORSA J. Comput.*, 1989.

[Gly89] P.W Glynn. A GSMP formalism for discrete-event systems. In *Proceedings of the IEEE*, volume 77, pages 14–23, 1989.

[Gv00] H. Gautama and A. J. C. van Gemund. Static performance prediction of data-dependent programs. In *Proceedings of the* 2^{nd} *International Workshop on Software and Performance*, pages 216–226, September 2000.

[HM04a] J. Hu and R. Mărculescu. Applicationspecific buffer space allocation for networksonchip router design. In *Proc. of the ICCAD*, 2004.

[HM04b] J. Hu and R. Mărculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of the Design Automation and Test in Europe Conference*, page 10234, 2004.

[HM05] J. Hu and R. Mărculescu. Energy and performance-aware mapping for regular NoC architectures. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(4), 2005.

[HMC97] S. Haddad, P. Moreaux, and G Chiola. Efficient handling of phase-type distributions in Generalized Stochastic Petri Nets. In 18^{th} *International Conference on Application and Theory of Petri Nets*, 1997.

[HN93] J. M. Harrison and V. Nguyen. Brownian models of multiclass queueing networks: Current status and open problems. *Queueing Systems*, 13:5–40, 1993.

[HZS01] X. S. Hu, T. Zhou, and E. H.-M. Sha. Estimating probabilistic timing performance for real-time embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):833–844, December 2001.

[Int93] International Organization for Standardization (ISO). *ISO/IEC 11172-3:1993 – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio*, 1993. http://www.iso.org/.

[Int05] International Telecommunication Union (ITU). *H.263 – Video coding for low bit rate communication*, 2005. http://www.itu.int/publications/itu-t/.

[JMEP00] R. Jigorea, S. Manolache, P. Eles, and Z. Peng. Modelling of real-time embedded systems in an object oriented design environment with UML. In *Proceedings of the* 3^{rd} *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC00)*, pages 210–213, March 2000.

[KJS+02] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, April 2002.

[KK79] P. Kermani and L. Kleinrock. Virtual Cut-Through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, 1979.

[Kle64] L. Kleinrock. *Communication Nets: Stochastic Message Flow and Delay*. McGraw-Hill, 1964.

[KM98] A. Kalavade and P. Moghé. A tool for performance estimation of networked embedded end-systems. In *Proceedings of the* 35^{th} *Design Automation Conference*, pages 257–262, 1998.

[Kop97] H. Kopetz. *Real-Time Systems*. Kluwer Academic, 1997.

[KRT00] J. Kleinberg, Y. Rabani, and E. Tardos. Allocating bandwidth for bursty connections. *SIAM Journal on Computing*, 30(1):191–217, 2000.

[KS96] J. Kim and K. G. Shin. Execution time analysis of communicating tasks in distributed systems. *IEEE Transactions on Computers*, 45(5):572–579, May 1996.

[KS97] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[LA97] Y. A. Li and J. K. Antonio. Estimating the execution time distribution for a task graph in a heterogeneos computing system. In *Proceedings of the Heterogeneous Computing Workshop*, 1997.

[Leh96] J. P. Lehoczky. Real-time queueing theory. In *Proceedings of the* 18^{th} *Real-Time Systems Symposium*, pages 186–195, December 1996.

[Leh97] J. P. Lehoczky. Real-time queueing network theory. In *Proceedings of the* 19^{th} *Real-Time Systems Symposium*, pages 58–67, December 1997.

[Lin98] Ch. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley and Sons, 1998.

[Liu94] D. Liu et al. Power consumption estimation in CMOS VLSI chips. *IEEE Journal of Solid-State Circuits*, (29):663–670, 1994.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):47–61, January 1973.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behaviour. In *Proceedings of the* 11^{th} *Real-Time Systems Symposium*, pages 166–171, 1989.

[LW82] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[Man04] S. Manolache. Fault-tolerant communication on network-on-chip. Technical report, Linköping University, 2004.

[MD04] S. Murali and G. De Micheli. Bandwidth-constrained mapping of cores onto NoC architectures. In *Proceedings of the Conference on Design Automation and Test in Europe*, 2004.

[MEP] S. Manolache, P. Eles, and Z. Peng. An approach to performance analysis of multiprocessor applications with stochastic task execution times. *Submitted for publication*.

[MEP01] S. Manolache, P. Eles, and Z. Peng. Memory and time-efficient schedulability analysis of task sets with stochastic execution time. In *Proceedings of the* 13^{th} *Euromicro Conference on Real Time Systems*, pages 19–26, June 2001.

[MEP02] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of multiprocessor real-time applications with stochastic task execution times. In *Proceedings of the* 20^{th} *International Conference on Computer Aided Design*, pages 699–706, November 2002.

[MEP04a] S. Manolache, P. Eles, and Z. Peng. Optimization of soft real-time systems with deadline miss ratio constraints. In *Proceedings of the* 10^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 562–570, 2004.

[MEP04b] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *ACM Transactions on Embedded Computing Systems*, 3(4):706–735, 2004.

[MEP05] S. Manolache, P. Eles, and Z. Peng. Fault and energy-aware communication mapping with guaranteed latency for applications implemented on NoC. In *Proc. of DAC*, 2005.

[MEP06] S. Manolache, P. Eles, and Z. Peng. Buffer space optimisation with communication synthesis and traffic shaping for NoCs. In *Proceedings of the Conference on Design Automation and Test in Europe*, March 2006.

[MP92] M. Mouly and M.-B. Pautet. *The GSM System for Mobile Communication*. Palaiseau, 1992.

[MR93] M. Malhotra and A. Reibman. Selecting and implementing phase approximations for semi-Markov models. *Stochastic Models*, 9(4):473–506, 1993.

[PF91] B. Plateau and J-M. Fourneau. A methodology for solving Markov models of parallel systems. *Journal of Parallel and Distributed Computing*, 12(4):370–387, 1991.

[PG98] J. C. Palencia Gutiérrez and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the* 19^{th} *IEEE Real Time Systems Symposium*, pages 26–37, December 1998.

[PKH01] E. L. Plambeck, S. Kumar, and J. M. Harrison. A multiclass queue in heavy traffic with throughput time constraints: Asymptotically optimal dynamic controls. *Queueing Systems*, 39(1):23–54, September 2001.

[PLB+04] M. Pirretti, G. M. Link, R. R. Brooks, N. Vijaykrishnan, M. Kandemir, and Irwin M. J. Fault tolerant algorithms for network-on-chip interconnect. In *Proc. of the ISVLSI*, 2004.

[PST98] A. Puliafito, M. Scarpa, and K.S. Trivedi. Petri Nets with k-simultaneously enabled generally distributed timed transitions. *Performance Evaluation*, 32(1):1–34, February 1998.

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.

[RE02] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. of DATE*, 2002.

[Ros70] S.M. Ross. *Applied Probability Models with Optimization Applications*. Holden-Day, 1970.

[SAN03] I. Saastamoinen, M. Alho, and J. Nurmi. Buffer implementation for proteo network-on-chip. In *Proceedings of the 2003 International Symposium on Circuits and Systems*, volume 2, pages II 113–II 116, 2003.

[Sch03] M. Schmitz. *Energy Minimisation Techniques for Distributed Embedded Systems*. PhD thesis, Dept. of Computer and Electrical Enginieering, Univ. of Southampton, UK, 2003.

[SGL97] J. Sun, M. K. Gardner, and J. W. S. Liu. Bounding completion times of jobs with arbitrary release times, variable execution times, and resource sharing. *IEEE Transactions on Software Engineering*, 23(10):604–615, October 1997.

[She93] G.S Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.

[SL95] J. Sun and J. W. S. Liu. Bounding the end-to-end response time in multiprocessor real-time systems. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, pages 91–98, April 1995.

[SN96] K. Shepard and V. Narayanan. Noise in deep submicron digital design. In *ICCAD*, pages 524–531, 1996.

[SS94] M. Spuri and J. A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, December 1994.

[Sun97] J. Sun. *Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed real-time systems. *Euromicro Jurnal on Microprocessing and Microprogramming (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[TDS+95] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W. S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 164–173, May 1995.

[TTT99] TTTech Computertechnik AG, TTTech Computertechnik AG, Schönbrunner Straße, A-1040 Vienna, Austria. *TTP/C Protocol*, 1999.

[van96] A. J. van Gemund. *Performance Modelling of Parallel Systems*. PhD thesis, Delft University of Technology, 1996.

[van03] A. J. C. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2003. to be published.

[Wil98] R. J. Williams. Diffusion approximations for open multiclass queueing networks: Sufficient conditions involving state space collapse. *Queueing Systems*, 30:27–88, 1998.

[YBD02] T.T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. In *Proc. of DAC*, pages 524–529, 2002.

[ZHS99] T. Zhou, X. (S.) Hu, and E. H.-M. Sha. A probabilistic performace metric for real-time system design. In *Proceedings of the* 7^{th} *International Workshop on Hardware-Software Co-Design*, pages 90–94, 1999.