Pre-Run-Time Scheduling of Distributed Real-Time Systems
Models and Algorithms

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op dinsdag 10 januari 1995 om 16.00 uur

door
Jacobs Philomena Cyprianus Verhoosel
geboren te Maastricht

druk: wibra dissertatie-drukkerij, helmond
Dit proefschrift is goedgekeurd door de promotoren
prof.dr.dipl.-ing. D.K. Hammer
en
prof.dr. L.R. Welch
en de copromotor
dr. F.J. Luit

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG
Verhoosel, Jacobus Philomena Cyprianus
Pre-run time scheduling of distributed real-time systems:
models and algorithms / Jacobus Philomena Cyprianus
Verhoosel. - Eindhoven : Eindhoven University of Technology
Thesis Eindhoven. - With index, ref. - With summary in Dutch.
ISBN 90-386-0414-9
Subject headings: pre-run-time scheduling / distributed real-time systems.
Veur Celine
Acknowledgements

I would like to thank the members of the DEDOS group for their valuable comments on presentations of my work. Stimulating discussions with Dieter Hammer, Lonnie Welch, Erik Luit, Alex Stoyenko, Klaus Ecker and Peter van der Stok have provided me with many new ideas.

I am also very thankful to Dieter Hammer, Alex Stoyenko, Lonnie Welch and Klaus Ecker for giving me many opportunities to meet colleagues from all over the world. Furthermore, I am obliged to Dieter Hammer, Erik Luit, Lonnie Welch, Emile Aarts and Klaus Ecker for reading a draft version of this thesis. They provided many suggestions for improvements.

Finally, I am very grateful to my family for supporting me during the past couple of years in many different ways. Especially, my wife Celine has always been a source of energy and encouragement.

Eindhoven, January 1995

Jack Verhoosel
# Contents

1 Introduction .......................... 1
   1.1 Guaranteeing Timeliness .......................... 1
   1.2 Pre-Run-Time Scheduling .......................... 4
   1.3 Thesis Overview .......................... 6

2 Scheduling Model .......................... 9
   2.1 Hardware Architecture Model .......................... 9
   2.2 Programming Model .......................... 12
      2.2.1 Abstract Data Type and Abstract Data Object Classes .......................... 12
      2.2.2 Processes .......................... 16
   2.3 Execution and Communication Paradigms .......................... 19
      2.3.1 Inter-processor Communication .......................... 21
      2.3.2 Method Execution .......................... 23
      2.3.3 Preemption and Beads .......................... 24
   2.4 Scheduling Constraints on Beads .......................... 28
      2.4.1 Absolute Timing Constraints .......................... 29
      2.4.2 Relative Timing Constraints .......................... 29
      2.4.3 Consistency Constraints .......................... 31
   2.5 Assignment and Scheduling Problems .......................... 31
      2.5.1 Process Assignment Problem .......................... 31
      2.5.2 Bead Scheduling Problem .......................... 35
   2.6 Assignment and Scheduling Approach .......................... 37
      2.6.1 Process Assignment .......................... 38
      2.6.2 Window Assignment .......................... 40
      2.6.3 Local Scheduling .......................... 48
   2.7 Conclusion .......................... 50
CONTENTS

3 Process Assignment .................................................. 53
  3.1 The Process Assignment Problem ......................... 53
  3.2 Problem Analysis ............................................. 54
    3.2.1 Bead Execution Intervals ......................... 54
    3.2.2 Expected Utilization .............................. 59
    3.2.3 Expected Parallelism .............................. 61
    3.2.4 Communication .................................. 62
  3.3 A Process Assignment Algorithm ......................... 64
    3.3.1 Communication Insertion ....................... 65
    3.3.2 Process Selection ............................... 67
    3.3.3 Processor Selection ........................... 68
    3.3.4 Process Reassignment ...................... 74
  3.4 Conclusion ............................................... 77

4 Window Assignment ................................................. 79
  4.1 The Window Assignment Problem ...................... 79
  4.2 Problem Analysis .......................................... 80
    4.2.1 Interval Adjustment to Correctness Constraints .. 85
  4.3 A Window Assignment Algorithm ....................... 89
    4.3.1 Consistency Constraint Transformation .......... 91
    4.3.2 Bead Selection .................................. 94
    4.3.3 Window Assignment .............................. 95
    4.3.4 Correct Interval Adjustment ................. 98
    4.3.5 Feasibility Condition Check ............. 103
  4.4 Conclusion ............................................... 104

5 Local Scheduling .................................................. 105
  5.1 The Local Scheduling Problem ...................... 105
  5.2 Problem Analysis .......................................... 108
    5.2.1 Feasibility Preserving Beads ............... 109
    5.2.2 Feasibility Preserving Bead Sequences .... 112
    5.2.3 Independent Partial Local Schedules ....... 116
  5.3 A Local Scheduling Algorithm ....................... 119
    5.3.1 Candidate Set Construction ............... 121
    5.3.2 Bead Selection ................................ 122
    5.3.3 Application of Theorems and Corollaries .... 124
    5.3.4 Backtracking ................................. 125
  5.4 Conclusion ............................................... 131
Chapter 1

Introduction

During the last decade, an increasing part of the functionality of systems in many application areas is handled by computers. As a consequence, the complexity of modern computer systems has increased proportionally and thus more effort is necessary to maintain the dependability of these computer systems. One of the dimensions in which the complexity of computer systems has grown is time. The environment of a computer system can impose certain real-time constraints on the tasks of the application. For instance, when the computer system in a nuclear power plant detects a melt down, it must in time initialize the insertion of safety bars into the nuclear kernel in order to prevent a major nuclear catastrophe. A computer system is dependable with respect to time, if all real-time constraints are met and thus timeliness is guaranteed.

1.1 Guaranteeing Timeliness

There are two scheduling approaches to achieve timeliness, each with its specific advantages and disadvantages: run-time scheduling and pre-run-time scheduling [17]. Systems that use run-time scheduling (also often called dynamic or on-line scheduling) usually operate in a non-predictable environment where new tasks can arrive at any moment. Such systems are very flexible and easily adaptable because everything is evaluated dynamically. In order to reduce
run-time overhead, run-time scheduling strategies usually use simple heuristic functions that can be efficiently evaluated. Two widely used run-time scheduling strategies for real-time systems are rate-monotonic scheduling and earliest deadline first scheduling. The rate-monotonic scheduling strategy assumes a set of independent periodic preemptable tasks on a single processor [26]. Each task has a certain execution time and must be executed once per period. The strategy ensures that the task with the smallest period among all available tasks is executed. Thereby, a task that is executing is preempted as soon as a task with a smaller period becomes available. The earliest deadline first strategy assumes a set of independent preemptable tasks where each task has a release time and a deadline [26]. Each task has a certain execution time and must be executed between its release time and its deadline. Thus, at each time, a number of tasks are ready to be executed. The strategy ensures that the task with the smallest deadline among all available tasks is executed. Thereby, a task that is executing is preempted as soon as a task with a smaller deadline becomes available.

The main disadvantage of run-time scheduling in general is that timeliness can not be guaranteed under all circumstances. Due to the resource commitments already made for current tasks, it might become impossible to accommodate new ones. For instance, simulation results of an evaluation of the run-time scheduling strategy described in [34] show that timeliness cannot be guaranteed under all circumstances [37]. Thus, timeliness can only be guaranteed under all circumstances when the environment is predictable. For a particular run-time scheduling strategy, it can be proven under which circumstances timeliness is guaranteed using this strategy. A good survey of run-time scheduling theory is given in [7]. Although run-time scheduling theory has received lots of consideration, the analyzed models are still restrictive and unrealistic for most applications.

For an arbitrary run-time scheduling strategy, the application can also be analyzed pre-run-time using schedulability analysis [39]. This technique checks whether timeliness is guaranteed for all possible run-time execution paths. The number of execution paths of an application depends on the number of conditionals in the code, because each conditional has several alternatives. In order to reduce the number of execution paths that have to be checked, several optimization techniques are used. A conditional can be transformed such that different alternatives become equivalent in the timeliness analysis [40]. Also, different conditionals can be linked when the selection criteria of the conditionals are mutually dependent [40]. The main disadvantage of schedulability analysis, however, is that all possible run-time
1.1. Guaranteeing Timeliness

execution paths have to be checked, which in general is an NP-complete problem.

In systems that use pre-run-time scheduling (also often called static or off-line scheduling), a schedule is constructed in advance for all tasks. This is done such that, at run-time all real-time constraints are satisfied and all shared resources are accessed without contention. Shared resources include processors, data, devices and communication media. In addition, pre-run-time schedules can be constructed for exceptional or faulty situations. If such a situation occurs at run-time the system can switch to an exceptional schedule that is also determined in advance.

Pre-run-time scheduling is especially useful for applications of which all the characteristics are known a priori and will not change very frequently, such as in factory automation, robotics, telecommunication, aerospace and copiers [11, 38]. For these applications, the amount of time needed to find a pre-run-time schedule is small relative to the average time between changes of the design.

Consider for instance a flexible production cell consisting of a moving belt, a robot, a plateau, and a production unit. The robot periodically takes an object from the moving belt and puts it on the plateau. The plateau turns and the production unit performs a number of operations on the object while still on the plateau. Meanwhile, the robot takes an already handled object from the plateau and puts it back on the moving belt. The production system is flexible in the sense that the production unit can perform different procedures on the objects. For each procedure a pre-run-time schedule can be constructed and loaded at run-time when necessary to handle an object. A straightforward design of this application consists of three periodic processes that manage the robot, the plateau and the production unit respectively. In addition, the three tasks defined by the processes have to be synchronized in time. The characteristics of this application are known before the system starts operating and the functionality only depends on the characteristics of the product and the production cell which do not change very frequently. Consequently, the construction of a pre-run-time schedule per product is appropriate.

A second example is a copier that consists of a paper input unit, an exposure part, and a paper output unit. As soon as the user gives the command to copy, the paper input unit takes an empty sheet of paper and puts it in the lighting position. Then, the exposure unit lights the sheet of paper thereby copying the page and the paper output unit puts the sheet of paper on an output plateau. For each possible copying task and exceptional situation, a
procedure exists that describes the steps to be taken by the copier. All these procedures are designed in advance and can thus be scheduled pre-run-time.

The main advantage of pre-run-time scheduling is that timeliness is guaranteed under all circumstances. The complete execution of the application is fixed before the system starts operating. Consequently, run-time scheduling becomes nothing more than dispatching tasks according to the constructed schedule and the run-time scheduling overhead is small. All synchronization requirements between tasks such as atomicity and relative timing constraints are also solved pre-run-time. Thus, there is no contention for any shared resource and run-time synchronization mechanisms, such as queuing at semaphores, are not necessary. Finally, the amount of preemption and thus the overhead of context switching at run-time can be controlled and reduced more easily when a schedule is constructed pre-run-time. The disadvantage of pre-run-time scheduling is that timeliness is guaranteed at the expense of low flexibility and adaptability. Pre-run-time scheduling is also very complex and uses worst-case, a priori information of the application which leads to low resource utilization.

In order to trade predictability against flexibility and good resource utilization, there is a trend to design systems that combine both methods [18]. In such systems, pre-run-time scheduling is used for the hard real-time part for which timeliness is absolutely necessary. This part usually contains not more than 10% of the tasks. Run-time scheduling is used for the remaining part of the system. This part mainly handles non-deterministic events by means of soft real-time tasks without strict real-time constraints. These tasks can then be scheduled and executed at run-time in the idle time intervals of the pre-run-time constructed schedule.

This thesis concentrates on pre-run-time scheduling and presents a novel, realistic scheduling model and a viable approach that can solve the resulting scheduling problem. A summary of the main results of this work can be found in Chapter 7.

### 1.2 Pre-Run-Time Scheduling

In general, scheduling problems are NP-complete, especially in distributed systems [16]. A consequence of the NP-completeness of a problem is that it is very hard\(^1\) to find an algorithm with a polynomial time-complexity

\(^{1}\)It has not yet been proven that it is impossible to find a polynomial algorithm for an NP-complete problem.
1.2. Pre-Run-Time Scheduling

that solves the problem. A large amount of literature concerns pre-run-time scheduling problems in non-real-time as well as real-time systems. A good survey of pre-run-time scheduling is given in [4]. The most comprehensive pre-run-time scheduling approaches for distributed real-time systems found in the literature are briefly described below. In each of the approaches, a real-time application basically consists of a set of periodic non-preemptable tasks with precedence relations. Additional constraints on the tasks and the system architecture on which the tasks have to be scheduled differ per approach.

In [14], the pre-run-time scheduling approach for the MARS system [23] is discussed. Applications for MARS have to be scheduled on a cluster of processors connected by a bus. Time slots of equal length on the bus are assigned to the tasks in a round robin fashion. A heuristic search strategy is presented that assigns the tasks to the processors and schedules task execution on the processors and message communication via the bus. The heuristic used is a probabilistic function of the message frequency. Besides the bus, no other resources are incorporated in this approach.

In [1], the same scheduling problem is considered. However, the messages via the bus are scheduled arbitrarily instead of in a round robin fashion as on the MARS bus. The paper describes a scheduling technique that resembles the approach in [42]. Both scheduling strategies construct on overall schedule by maintaining a partial local schedule for each processor. Repeatedly a task is assigned to a processor and the partial local schedules are adjusted accordingly.

In [51], each task in a real-time application has an additional release time and deadline and exclusion relations are imposed on tasks whose execution may not overlap in time. An application has to be feasibly scheduled on a set of identical processors. Thus, a schedule has to be found in which each task executes between its release time and deadline and all precedence and exclusion relations are satisfied. The paper does not consider the communication network between the processors as a separate resource. The scheduling algorithm starts with an initial schedule that is generated using an extended version of the earliest deadline first strategy. Then, the initial schedule is repeatedly improved until a feasible schedule is found. Results of experiments with tasks sets of up to 420 tasks on 2 to 3 processors are presented.

Finally, in [33], an application is assigned to and scheduled on a distributed system. Each task can have resource requirements that restrict the assignment of the task to only those processors that can access the resource. A pre-run-time scheduling algorithm is presented that incorporates commu-
Chapter 1. Introduction

The algorithm first decides whether a cluster of communicating tasks should be assigned to the same processor. This is done using a heuristic that is based on the computation times of the tasks in the cluster and the amount of communication between them. Then, clusters of tasks are assigned to processors and a schedule that incorporates communication is determined again using heuristics.

1.3 Thesis Overview

In Chapter 2 of this thesis, a novel pre-run-time scheduling model for distributed real-time systems is described. The hardware architecture of a distributed real-time system consists of processors, connected by a hierarchical network, and devices such as sensors and actuators. An application is designed according to an object-based programming paradigm and therefore consists of objects that can privately manage resources such as data and devices. The objects are assumed to be passive and are clustered into processes that initialize several concurrent threads of control for the application. Such a thread can only be preempted at pre-defined preemption points such as remote procedure calls. Each bead represents the non-preemptable execution of a piece of method code between two successive preemption points. The dynamic structure of an application is therefore modeled as a graph of beads. The model defines several constraints on the beads of an application. Absolute timing constraints represent periodicity and deadlines, relative timing constraints model several kinds of timed precedence relations within a thread of control or between several threads of control, and consistency constraints enforce consistent use of shared resources.

Based on this model, Chapter 2 also presents a pre-run-time assignment and scheduling approach that can solve the resulting pre-run-time scheduling problems. The approach consists of three steps. In the first step, the processes of an application are assigned to the processors of a distributed system and communication beads are inserted. In the second and the third step, each bead is scheduled on the processor on which it executes. Therefore, in the second step, an execution interval, called window, is assigned to each bead. This is done such that all constraints are satisfied when each bead is scheduled in its window. In the third step, the beads at each processor are scheduled locally such that each bead executes in its window and no two beads at the same processor overlap in time. The main advantage of the window approach is that the scheduling problem at each processor can
be solved in isolation.

Three constructive heuristic algorithms are presented that can solve the scheduling problems of each step. A heuristic algorithm uses problem-related “rules of thumb” that can be easily calculated in order to find an approximate, non-optimal solution in “reasonable time” instead of finding an optimal solution with much higher effort. However, there are always problem instances for which the algorithm cannot find a solution in “reasonable time”, even if one exists. This is inherent to an NP-complete problem and thus every algorithm that attacks such a problem suffers from this disadvantage.

Chapter 3 attacks the problem of assigning the processes of an application to the processors of a distributed hardware architecture. This has to be done such that the number of devices managed by the objects assigned to a particular processor does not exceed an upper bound. A heuristic algorithm is presented that can solve this problem. For each process, a processor is selected based on a novel objective function that weights the amount of communication against the amount of parallelism. If a processor can not be found, the algorithm reassigns processes and makes another attempt. When processes are assigned to processors, the algorithm inserts communication beads into the execution graph where necessary.

Chapter 4 discusses the problem of assigning a window to each bead. Windows have to be assigned such that all constraints are satisfied when each bead is scheduled in its window. The heuristic algorithm presented in this chapter can solve the problem, assigns a window to each bead such that two feasibility conditions are satisfied. It first transforms consistency constraints into relative timing constraints. Then, a window is assigned to each bead by a heuristic that attempts to spread the execution time evenly over the most critical path. After the assignment of a window to a bead, the windows of other beads are adjusted or shifted such that the feasibility conditions can still be satisfied.

Chapter 5 describes the problem of scheduling a set of beads with windows on a single processor. Thus, each bead has to be scheduled in its window and no two beads may overlap in time. A heuristic algorithm is presented that can solve this problem. The algorithm constructs a local schedule by repeatedly adding beads to the end of a partial feasible local schedule. Bead selection is done using an enhanced version of the earliest deadline first strategy that is based on three theorems that describe certain characteristics of the problem. If the partial feasible local schedule cannot be feasibly extended with a new bead, the algorithm uses intelligent back-
tracking heuristics. Thereby, certain beads at the end of the partial feasible local schedule are replaced by more promising ones.

In Chapter 6, results of performance measurements of all three algorithms in isolation and of the entire pre-run-time scheduling approach are presented and discussed.

Finally, the main conclusions of the work presented in this thesis are summarized in Chapter 7 and suggestions for future research are described.
Chapter 2

Scheduling Model

In this chapter, a model for pre-run-time scheduling of distributed real-time systems is given. The model consists of a hardware architecture model, a programming model, and execution and communication paradigms. The hardware architecture model describes the hardware of the distributed real-time system. The programming model describes how applications have to be programmed and the execution and communication paradigms describe how applications are executed on the hardware architecture. There are several constraints on the application. Absolute timing constraints represent periodicity and deadlines. Relative timing constraints model several kinds of timed precedence relations and timed synchronization requirements. Consistency constraints enforce consistent use of data and devices. Additionally, the model is formalized to obtain a mathematical foundation on which assignment and pre-run-time scheduling problems are defined. More details of the pre-run-time scheduling model can be found in [47]. Finally, an approach is presented that can solve these assignment and pre-run-time scheduling problems.

2.1 Hardware Architecture Model

The hardware architecture of a distributed computer system contains several processors modeled as a set of processors \( P/R \) (see Figure 2.1). Processors are homogeneous and thus have the same execution speed. Each processor has
a piece of private memory that can only be accessed by the processor itself. In private memory a part of the application can be stored. The amount of private memory of a single processor is assumed to be large enough to store any application. Thus, the private memory size is not a constraining factor for assignment of the application to the distributed system. Therefore, private memory is not part of the hardware architecture model.

Different processors communicate via a network modeled as a set of communication media \( CM \). A communication medium is assumed to be either a bus that typically connects two or more processors or a high-speed bidirectional point-to-point link that connects two processors directly. The processors connected by a bus have to contend to get access to the bus. On the other hand, the two processors connected by a link do not contend for the link, because they always communicate with each other. The bus concept can model communication media such as a VME or VSB bus which are commonly used as an internal bus in a VME shared memory system. The definition of a link allows the modeling of all kinds of networks that consist of a set of those direct links, such as an Ethernet LAN, a token-ring LAN, a hypercube, a mesh, or a WAN. Each communication medium \( cm \in CM \)
has an arbitrary bandwidth \( cm.bw \in \mathbb{N}^+ \) that limits the rate at which data can be transmitted. In addition, delay \( cm.bd \in \mathbb{N}^+ \) is the time needed to transport a piece of data of bandwidth size \( cm.bw \) via medium \( cm \). The way in which the delay of a communication of a piece of data of arbitrary size via medium \( cm \) depends on \( cm.bw \) and \( cm.bd \) is discussed in more detail in Section 2.3.1.

The only assumption that is made about the communication media interconnection topology among processors is that each pair of processors is physically connected via the network. A processor communicates with another processor via a uni-directional route of busses, links and processors. A route is defined as a sequence of hops, where each hop is a triple \((sp, cm, rp)\) of a sending processor \( sp \in PR \), a communication medium \( cm \in CM \) and a receiving processor \( rp \in PR \).

**Definition 2.1 (Route)** A route \( r_{oi,j} \) from processor \( pr_i \) to processor \( pr_j \) is a sequence \( \{(sp_1, cm_1, rp_1), ..., (sp_n, cm_n, rp_n)\} \) of \( n \) hops for which (1) \( sp_1 = pr_i, rp_1 = pr_j \) and (2) \( (\forall k : 1 \leq k < n : rp_k = sp_{k+1}) \) holds.

In a route, no distinction is made between busses and links. Note that a bus and a link can be part of different routes. Thus, communications via different routes may contend for the same medium. The set \( RO \) contains a route from each processor in \( PR \) to each other processor in \( PR \).

Different processors can synchronize via global data in *shared memory* modules. A shared memory module is assumed to be accessible by a group of processors. In Section 2.3.1, it is described how shared memory can also be used for communication between processors. The amount of shared memory is assumed to be large enough to store global data of any application. Thus, the shared memory size is not a constraining factor for assignment of the application to the distributed system. Therefore, shared memory is not part of the hardware architecture model.

Processors can interact with the system environment via physical *devices* such as sensors, actuators, and disks, that are modeled as a set of devices \( DV \). In order to be predictable, each device has a bounded access time. An access to a disk usually involves large amounts of data. In order to keep disk access times relatively small, real-time disks are assumed that utilize a special storage scheme. Access times for sensors and actuators are relatively small, because only a small amount of data is involved. The way in which access times are actually calculated is discussed in Section 2.3.2.

It is assumed that the device configuration is *not* fixed until an application is assigned to the architecture. Thus, each device still has to be
connected to a particular processor. In Section 2.2, it is discussed how devices are managed by an application and that the assignment of the devices to the processors is known after the assignment of the application to the processors. After assignment of a device to a processor, the device can only be accessed via that processor. Finally, only a maximum number of devices \( pr_{md} \in N^+ \) can be connected to a processor \( pr \in PR \), due to physical constraints. Since processors are homogeneous, \( pr_{md} \) is equal for all processors \( pr \).

### 2.2 Programming Model

The increasing complexity of modern computer systems necessitates the development of programming methods that allow the user to manage a high degree of complexity in an easy way. Object-based methods are well suited for this purpose [35, 50], due to their convenient structuring possibilities. Therefore, this thesis focuses on applications designed according to an object-based programming paradigm. Additional features of the conventional object-oriented programming paradigm, such as inheritance and dynamic binding, are not discussed in this thesis. However, the scheduling model is independent of these features. Throughout the remaining of this chapter several concepts are exemplified by an object-based design of a small control subsystem for a particle accelerator. This subsystem is part of a larger system described in more detail in [38].

#### 2.2.1 Abstract Data Type and Abstract Data Object Classes

One of the main advantages of an object-based programming model is that a designer can structure an application around abstractions used in the application. Furthermore, a well-structured application consists of loosely-coupled objects that are very well suited for reuse. The basic concept in an object-based programming paradigm is the class concept. A class is a basic building block that can be instantiated in order to build an application. A class is either an abstract data type or an abstract data object.

**Definition 2.2 (Abstract data type)** An abstract data type (ADT) defines and exports (1) a type that can be used to declare variables and (2) a set of methods to manipulate variables of the provided type. □

In Figure 2.2a, an example ADT `Disk.Manager` is specified that defines the type `Disk` and three methods to write, read and clear variables of type
2.2. Programming Model

Disk. The write method writes a string \( r \) to a disk \( d \). The read method takes

\[
\text{ADT Disk:\text{Manager;}} \\
\text{TYPE Disk;} \\
\text{METHOD Write:\text{Disk}} \\
\hspace{1cm} \text{(in } d:\text{ Disk; in } r:\text{ String);} \\
\text{BEGIN} \\
\hspace{1cm} pp_d.1 \rightarrow \text{ "write string } r \text{ to disk } d\text{";} \\
\hspace{1cm} \text{END METHOD;} \\
\text{METHOD Read:\text{Disk}} \\
\hspace{1cm} \text{(in } d:\text{ Disk; out } r:\text{ String);} \\
\text{BEGIN} \\
\hspace{1cm} pp_d.2 \rightarrow \text{ } r \text{ := "data on disk } d\text{";} \\
\hspace{1cm} \text{END METHOD;} \\
\text{METHOD Clear:\text{Disk}} \hspace{1cm} \text{(in } d:\text{ Disk);} \\
\text{BEGIN} \\
\hspace{1cm} \text{WHILE } \text{ "used sectors left on } d\text{"} \\
\hspace{2cm} \text{DO } \{0:\frac{1}{4}; 1:\frac{1}{2}; 2:\frac{3}{4}\} \\
\hspace{1cm} \text{pp_d.3 } \rightarrow \text{ "clear used sector on } d\text{";} \\
\hspace{1cm} \text{END WHILE;} \\
\text{END METHOD;} \\
\text{END ADT.;}
\]

(a)

\[
\text{ADO Sensor;} \\
\hspace{1cm} \text{DATA value: Real;} \\
\text{METHOD Read:\text{Sensor}} \\
\hspace{1cm} \text{(out } v:\text{ Real);} \\
\text{BEGIN} \\
\hspace{1cm} pp_s.1 \rightarrow \text{ } v \text{ := value;} \\
\hspace{1cm} \text{END METHOD;} \\
\text{END ADO;} \\
\]

(b)

Figure 2.2: Specification of an ADT (a) and an ADO (b).

a piece of data from disk \( d \) and assigns this to string \( r \). The clear method removes all data from a disk \( d \). In addition, selection probabilities for the while statement of this method are specified. This is discussed in detail below. Finally, the markers with arrows depicted left from the program text indicate preemption points. This is discussed in detail in Section 2.3.3.

The instantiation of an ADT is called a facility. A facility contains and exports the defined type and the defined set of methods. After instantiation of an ADT, multiple variables can be declared of the type exported by the facility. The methods exported by a facility can manipulate all variables declared of the type that is exported by the facility. Figure 2.3a visualizes how ADT Disk:\text{Manager} is instantiated into a facility \( DM \) and how two variables \( D_1 \) and \( D_2 \) are declared of type Disk. To support the development of loosely-coupled, highly cohesive, reusable program units, Ada, Chu, Modula-2 and RESOLVE provide the ADT construct [9, 25, 31, 48].
Figure 2.3: A facility DM instantiated from ADT Disk_Manager that contains two disks \( D_1 \) and \( D_2 \) of type Disk (a) and an object \( S \) instantiated from ADO Sensor that encapsulates a piece of data value (b).

Definition 2.3 (Abstract data object) An abstract data object (ADO) defines (1) a data part and (2) exports a set of method to enable the manipulation of this data.

While the ADT is primarily used for defining and exporting types, the ADO is used to encapsulate data. The encapsulated data is accessible only by calling one of the methods exported by the object. In Figure 2.2b, an example ADO Sensor is shown which defines the value of a sensor and exports a method to read this value. The instance of an ADO is called an object. An object encapsulates a data value and exports a set of methods to manipulate this data. Figure 2.3b visualizes how the ADO Sensor is instantiated into an object \( S \). The ADO construct is supported in languages such as RT-Euclid, C++, Eiffel and Smalltalk [22, 30, 32, 41]. Similar to an ADO, an ADT may also define a data part. Therefore, an ADO is a special case of an ADT, namely one that exports no type and that defines data. Thus, ADOs can also be developed in Ada, Clu, Modula-2 and RESOLVE.

One of the similarities between ADTs and ADOs is that the type defined by an ADT as well as the data defined by an ADO either represent (1) a kind of device, such as sensor, actuator, or disk or (2) a kind of data structure
used in the application. In the first case, all variables declared of the device type and the data encapsulated by a device object represent abstractions of physical devices. Consequently, a facility can privately manage several physical devices of the same kind, while an object can privately manage only a single physical device. In order to facilitate access to the device, it must be assigned to the processor to which the facility or object managing it is assigned.

A major distinction between an ADO and an ADT is that an ADO yields a data value upon instantiation, while an ADT requires variables to be explicitly declared following instantiation. Additionally, the methods of a facility usually manage *multiple* variables of the exported type. For instance, the methods of facility $DM$ manage both disks $D_1$ and $D_2$ of type Disk exported by the facility (see Figure 2.3a). In contrast, the methods of an object manage a *single* piece of data (see Figure 2.3b). This difference between an ADO and an ADT becomes important when processes constructed from ADTs and ADOs are executed in parallel. Clients that want to manipulate different variables of a type exported by a facility use the same set of methods. Hence, these clients contend for the same processor, because all the methods of a facility are assigned to the same processor. This contention can easily be avoided by replication of the facility, where the variables of the facility are distributed among the replicates. If the replicates are assigned to different processors, the clients can manipulate the variables without contention. Replication of an object is also possible. In this case, each replicate contains a copy of the same piece of data. However, there is an additional overhead to maintain the same value in all copies.

Within a method of a facility or object, all types of statements can be used. Besides ordinary assignments, the designer is allowed to use structured statements, such as a conditional (if-statement) and repetitions (while-statement), as well as calls to other methods. In order to be predictable, each statement must have a *bounded* execution time. Therefore, (indirect) recursion is prohibited, because the depth of the recursion is hard to limit. These assumptions do not restrict the expressive power of the programming language because recursion can be implemented by repetitions. However, the designer is required to specify the maximum number of times the body of a repetition is executed. This guarantees a bounded execution time of the repetition. In addition, the designer is allowed to specify a *selection probability* for each alternative of a conditional and for each number of iterations of a repetition. For instance, the maximum number of times the body of the while-statement in method Clear_Disk can execute equals 2 (see Figure 2.2a).
Chapter 2. Scheduling Model

The probability that the body of the while-statement is not executed is $\frac{1}{3}$, the probability that the body of the while-statement is executed once is $\frac{1}{2}$, and the probability that the body of the while-statement is executed twice is $\frac{1}{6}$. It is assumed that these probabilities are derived from statistical data of the particular application. If the selection probabilities for a conditional or repetition are not given, it is assumed that each alternative or number of iterations has an equal selection probability. The selection probabilities are used during assignment to assess processor utilization.

2.2.2 Processes

In order to facilitate the design process, the designer has access to a library of pre-defined reusable ADTs and ADOs from which an application can be build. An application consists of a set of processes.

**Definition 2.4 (Process)** A process is a piece of code that (1) creates facilities and objects by instantiating pre-defined ADTs and ADOs, (2) may define its own facilities and objects, and (3) may have a body that defines the functionality of the process as a sequence of method calls.

Each facility and object created by a process is either instantiated from a pre-defined ADT or ADO, or defined by the process itself. The latter enables the designer to program application-specific functionality. The body of a process may be empty. In this case, the process is passive and waits for a call to a method of one of its facilities and objects.

The way in which an application is divided into different processes, facilities and objects, depends on the characteristics of the application. However, for the applications considered in this thesis it is assumed that the number of processes is at least equal to the number of processors. Another important assumption is that a process and all the facilities and objects that it creates have to be assigned to the same processor. In the remainder, the term process is therefore used to indicate a process together with all its facilities and objects. The assumption is based on the notion that there is usually a lot of interaction between the facilities and objects within the same process. Hence, the amount of communication is reduced when these facilities and objects are assigned to the same processor. Furthermore, the search space of the problem to assign an application to a hardware architecture is reduced significantly, because the number of processes in an application is usually many times smaller than the number of facilities and objects.
2.2. Programming Model

A process may export its facilities and objects, while another process can import an exported facility or object. Thereby, the methods of the imported facility or object become available to the importing process and all its facilities and objects. A process does not manage variables or encapsulate data directly. However, it can manipulate these indirectly by performing a method call to the facility managing the variable or the object encapsulating the data.

The example control subsystem of a particle accelerator, given in Figure 2.4, controls the sensing, archiving and reporting of the detection of two kinds of particles: electrons and protons. A sensor that detects electrons and protons is attached to the accelerator tube. Detections of electrons and protons are registered on two separate disks respectively. In addition, the system periodically generates a report that contains information about electron and proton detections. A straightforward design of the subsystem therefore consists of three processes.

Process Archiving instantiates a facility DM of ADT Disk.Manager (see Figure 2.2), and declares two variables ED and PD of type DM.Disk that represent the two disks. The facility as well as both disks are exported to other processes. The body of process Archiving simply consists of two method calls to clear disks ED and PD.

Process Sensing imports facility DM as well as both disks ED and PD from process Archiving. Additionally, the process instantiates an object S from ADO Sensor (see Figure 2.2), and defines an object PS that encapsulates a piece of data called energy that represents the latest value detected by sensor S. Object S is marked as delimited to introduce new preemption points. This is discussed in detail in Section 2.3.3. Object PS contains a single method Sense.Particle. This method reads the energy value from the sensor, determines the particle type, if any, based on the energy and records this on the appropriate disk. The selection probabilities of the conditionals in the method are such that the probability that an electron is detected equals \( \frac{1}{10} \) and the probability that a proton is detected equals \( \frac{9}{10} \). The body of process Sensing consists of a single call to method Sense.Particle.

Process Reporting imports facility DM as well as both disks ED and PD from process Archiving. Additionally, the process defines an object RG that encapsulates the report latest generated. Object RG contains a single method Generate.Report that reads an yet unread piece of particle detections registered on disk d and sends a report to the user. The body of process Reporting consists of two method calls to generate reports for disks ED and PD.
PROCESS Sensing;
IMPORTED FACILITY
DM = Archiving.DM;
IMPORTED VARIABLE
ED = Archiving.ED;
PD = Archiving.PD;
OBJECT S: Sensor;
     (Delimited)
     - pre-defined -
OBJECT PS: Particle-Sensing;
DATA energy: Real;
METHOD Sense_Particle;
BEGIN
  pp₄,₁ ← S.ReadSensor(energy);
  IF (energy = "electron-energy")
    THEN (1/10)
    pp₄,₂ ← DM.Write.Disk
             (ED, "(electron.time)");
  ELSE (1/5)
    IF (energy = "proton-energy")
      THEN (1/10)
      pp₄,₃ ← DM.Write.Disk
               (PD, "(proton.time)");
  END IF;
END IF;
END OBJECT;
pp₄,₁ ← BEGIN (Period = 50)
  PS.Sense_Particle;
pp₄,₃ ← END PROCESS;

PROCESS Archiving;
IMPORTED FACILITY
DM = Archiving.DM;
IMPORTED VARIABLE
ED = Archiving.ED;
PD = Archiving.PD;
OBJECT RG: Report-
     Generating;
     DATA report: String;
     METHOD Generate_Report
     (in d: DM.Disk);
BEGIN
  pp₃,₁ ← DM.Read.Disk(d, report);
  "send report to display";
END METHOD;
END OBJECT;
pp₃,₁ ← BEGIN (Period = 50,
                 Deadline = 40)
  RG.Generate_Report(ED);
  RG.Generate_Report(PD);
pp₃,₃ ← END PROCESS;

Figure 2.4: A design of the control subsystem of the particle accelerator.
The last feature of the programming model is that the designer is allowed to impose timing constraints on the application. In particular, each process of an application executes periodically starting at relative time 0. The designer can specify an appropriate period for each process. For example, all processes in the control subsystem have a period of 50 time units (see Figure 2.4). The periods of the processes in the control subsystem are not very realistic, because in reality process Sensing executes far more often than processes Archiving and Reporting. However, it is easier to explain the example when all periods are chosen to be equal. The designer can also impose a deadline on an execution of the body of a process. This deadline is relative to the start of a period and has to be smaller than the period. For instance, deadline 40 is imposed on the execution of the body of process Reporting. Finally, besides periods and deadlines, the designer is also allowed to impose other kinds of timing constraints that relate executions of different processes (see Section 2.4).

Summarizing, the static structure of an application consists of a set of processes that privately manage devices. This static structure is modeled by a set of processes $PS$. Each process $ps \in PS$ has an attribute $ps.nd \in \mathbb{N}$ that indicates the number of devices privately managed by $ps$. The static structure of the control subsystem is modeled as:

\[
PS = \{ \text{Sensing, Archiving, Reporting} \} \\
\text{Sensing}.nd = 1, \text{Archiving}.nd = 2, \text{Reporting}.nd = 0.
\]

In addition to a set of processes that models the static structure of an application, there is a constraint on the assignment of the processes to the hardware architecture. This constraint expresses that the total number of devices implicitly assigned to a processor $pr$ may not exceed the maximum number of devices $pr.md$ that can be connected to the processor.

### 2.3 Execution and Communication Paradigms

In the previous two sections, the hardware architecture model and the programming model are presented. Now, it is possible to describe how an application is actually executed on a distributed system. First of all, each process of the application has to be assigned pre-run-time to a particular processor of the distributed system. When the system is initialized, each process is loaded on the processor to which it is assigned and each process creates all its facilities and objects. Then, each process with a body initializes a so-called
activity by starting to execute the sequence of method calls in the body [18].

Definition 2.5 (Activity) An activity is the flow of control defined by the sequence of method calls in the body of a process, including all nested method calls and all possible execution paths. The activity initialized by a process is periodic and inherits the period and optional deadline of the process.

Due to the export and import constructs, activities can switch execution from one process to another process (see Section 2.2.2) even at another processor. In addition, an activity can take one of several possible execution paths due to conditionals and repetitions within a method. For instance, process Sensing defines an activity that starts with method Sense_Particle of object PS that contains (1) a nested call to method Read_Sensor of object S and (2) a conditional that enables a nested call to method Write_Disk of facility DM with either disk ED or PD as parameter. This activity is executed periodically with a period of 50 time units. Process Archiving defines an activity that periodically executes method Clear_Disk of facility DM for disks ED and PD. The period of this activity is 50 time units. Process Reporting defines an activity that periodically executes method Generate_Report of object RG for disks ED and PD. The period of this activity is 50 time units, but it also has a deadline of 40 time units. Thus, the set of processes of an application defines a set of activities that are executed concurrently.

Definition 2.6 (Execution) An execution of an application is the set of concurrent activities defined by the processes of the application.

When an activity performs a method call or returns from a method call, parameter data has to be passed. For instance, when _DM.Clear_Disk(ED) is called in the body of process Archiving, the actual parameter ED has to be passed to facility DM to replace the formal parameter d of method Clear_Disk.

With respect to where facilities and objects are created, two different kinds of method calls are distinguished.

Definition 2.7 (Local procedure call) A local procedure call (LPC) is a call to a method of a facility or object in the same process.

For instance, the call _S.Read_Sensor(energy) in object PS created in process Sensing is an LPC, because objects S and PS are created in the same process. Since a process is the unit of assignment to processors, the caller and
2.3. Execution and Communication Paradigms

callee in an LPC are assigned to the same processor. An LPC is therefore implemented as a synchronous call that passes the parameter data as usual via the stack.

**Definition 2.8 (Remote procedure call)** A remote procedure call (RPC) is a call to a method of a facility or object in another process.

For instance, the call $DM.\text{Read.Disk}(d, \text{report})$ in object $RG$ created in process Reporting is an RPC, because facility $DM$ is created in process Archiving. An RPC is a synchronous call and thus the caller blocks on the call and only continues after the return of the call. In contrast to an LPC, the caller and callee in an RPC may be assigned to different processors. In this case, parameter data is passed back and forth by communication via the routes between the two processors involved.

### 2.3.1 Inter-processor Communication

Each communication medium (bus or link) is assumed to have a fixed communication mode that is either synchronous or asynchronous. Therefore, each medium $cm \in CM$ has an attribute $cm.\text{mod}$ that indicates the communication mode of $cm$.

For an inter-processor communication via an asynchronous medium, it is assumed that there is a communication mechanism that separates in time the sending and receiving phase of the communication. During the communication, some intermediate buffer is used (see Figure 2.5a). For a bus

![Diagram](attachment:image.png)

**Figure 2.5:** Inter-processor communication in asynchronous mode (a) and synchronous mode (b).

communication this can be, e.g., a mailbox in shared memory [10]. For a link
communication, this can be a piece of private memory at the sending or receiving processor. In the sending phase of an asynchronous communication, a processor sends (1) the data from its private memory to the buffer and (2) the pointer to the data to the buffer control mechanism. In the receiving phase of such a communication, a processor receives (1) the pointer from the buffer control mechanism and (2) the data from the buffer and stores this data in private memory. The sending as well as the receiving phase consists of repeatedly transporting successive pieces of bandwidth size data to or from the buffer. The advantages of asynchronous communication are that it allows the receiving processor to receive the data at any time after it has been put into the buffer and that several communications can be buffered.

Inter-processor communication via a synchronous medium is achieved under strict synchronization of the sending and receiving processor (see Figure 2.5b). The sending processor first signals the receiving processor via the medium that it wants to send a specific amount of data. When this signal is received, the receiving processor reserves a piece of private memory of sufficient size. The sending processor then transports successive pieces of bandwidth size data via the medium, while at the other end the receiving processor receives the data and stores it in private memory. Consequently, there is no distinction between the sending and receiving phase.

The time needed to transport data on a medium is determined by the amount of data, and the bandwidth and bandwidth size delay of the medium. A communication of a piece of data of size $ds$ via an asynchronous medium $cm$ consists of (1) $[ds/cm.bw]$ cycles to transport a bandwidth size piece, followed by (2) transport of the pointer. The pointer is assumed to be much smaller than the bandwidth size. Note that both phases of an asynchronous communication require this number of cycles. A communication of a piece of data of size $ds$ via a synchronous medium $cm$ consists of (1) an initialization signal, followed by (2) $[ds/cm.bw]$ cycles to transport a bandwidth size piece. The initialization signal is assumed to be much smaller than the bandwidth size. Consequently, communication of data of size $ds$ via an asynchronous or synchronous medium $cm$ requires $t_{cm,ds} = ([ds/cm.bw] + 1) \times cm.bd$ time units. All these delays assume that there is no contention for the communication medium, which is a reasonable assumption because each medium is scheduled explicitly pre-run-time.
2.3. Execution and Communication Paradigms

2.3.2 Method Execution

During the execution of a method of a facility or object that manages a device, the device can be physically accessed. Two kinds of device accesses are distinguished: a non-blocking device access and a blocking device access. A non-blocking device access strongly resembles an access to a data structure managed by an object. In such a device access, the processor that executes the method writes data to or reads data from a device register in a few processor cycles. Thus, the access time is relatively small and bounded. For instance, writing a new instruction to an actuator can be represented by this kind of access. Consequently, during a non-blocking device access, the processor is not idle and the activity that performs the access is not blocked.

On the other hand, an activity can get blocked during a device access, e.g., when reading from or writing to a disk, or reading the value of a sensor. Such a blocking device access globally consists of three phases (see Figure 2.6). First, the processor activates the device by writing relevant data into the device registers. This is the initialization phase. Second, the device performs the physical access and sends a device interrupt to the processor. This is the access phase. During the access phase, the activity is blocked and the processor can continue with other activities. Finally, the processor reads the result of the access and the activity continues. For instance, in each method of ADT DiskManager, a blocking device access to disk d is performed (see Figure 2.2a). In addition, during the assignment statement \( v := value \) in method ReadSensor of ADO Sensor, a blocking device access to the sensor is performed (see Figure 2.2b).

During the second phase of a blocking device access, the activity that performs the access is blocked. It is assumed that the blocking time of a device is bounded and consists of two parts: (1) a constant blocking time that the device always needs to perform the access and (2) a variable blocking time that depends on the amount of data that is involved in the access. For
instance, writing a piece of data to a disk consists of positioning the head of the disk followed by several cycles of writing a piece of data of unit size. Hence, for each device $dv \in DV$, a constant blocking time $dv.cb \in \mathbb{N}$ is defined. The variable blocking time of an access in which data of unit size is involved, is denoted by a worst case time $dv.vb \in \mathbb{N}$. Finally, the blocking time of a blocking device access to a device $dv$ in which data of size $ds$ is involved, is defined as $\alpha_{dv,ds} = dv.cb + (ds \times dv.vb)$.

### 2.3.3 Preemption and Beads

Scheduling models typically assume that dynamic entities, such as tasks, modules, procedures, segments, and so on, are either preemptable at any point or not preemptable at all. Unrestricted preemptability may result in an excessive amount of overhead due to context switching, while non-preemptability decreases the schedulability. Therefore, in this thesis a trade-off is made between these two extremes and it is assumed that activities are semi-preemptable. This means that preemption is only allowed at specific pre-defined preemption points. Three basic types of preemption points are defined:

1. At an RPC and the return from an RPC, because blocking communication might be involved. For instance, call $DM.Write.Disk(EP,...)$ from process Sensing to process Archiving causes two preemption points $pp_{s,4}$ and $pp_{s,5}$ at the call and the return respectively (see Figure 2.4).

2. Between the sending and receiving phase of an asynchronous inter-processor communication (see Figure 2.5a). In this figure, the two small circles connected by the dashed arrow enclose the preemption point. Thereby, the receiving processor is enabled to receive the data at any time after it has been sent. Obviously, these preemption points are only known when the processes are assigned and communication is inserted.

3. At the access phase of a blocking device access (see Figure 2.6). In this figure, the two small circles indicate the start and end of the access phase and enclose the preemption point. This is a natural preemption point, because the activity becomes blocked and the processor can continue with other activities. For instance, writing a string to a disk in AIDT Disk.Manager causes a preemption point $pp_{d,1}$ (see Figure 2.2a).
2.3. Execution and Communication Paradigms

In addition, the designer can insert additional preemption points in order to control the number of context switches. He can mark facilities and objects as delimited and thereby define a fourth type of preemption point:

4. At a delimited LPC and the return from a delimited LPC.

For instance, object $S$ in process Sensing is marked as delimited. Thus, the LPC $S$.Read_Sensor(energy) in object $PS$ and the return from this LPC cause preemption points $pp_{a,2}$ and $pp_{a,3}$ (see Figure 2.4). The designer can even make all calls and returns preemption points, by marking all facilities and objects as delimited.

In order to perform assignment and pre-run-time scheduling, a deterministic pre-run-time view of activities has to be derived that incorporates all possible execution paths. This pre-run-time view is based on the predefined preemption points. In order to include all possible execution paths pre-run-time, the control flow has to be determined from the program code. In the pre-run-time view, a preemption point can thus have several successor preemption points due to conditionals and repetitions. For instance, the if-then-else statement in method Sense_Particle of object $PS$ contains a preemption point in the then-clause as well as the else-clause. Therefore, according to the control flow, preemption point $pp_{a,3}$ at the return of the delimited LPC $S$.Read_Sensor(energy) has three successor preemption points: $pp_{a,4}$ and $pp_{a,6}$ for each RPC $DM$.Write_Disk and $pp_{a,8}$ at the end of the activity that is started by process Sensing. Also, the while-statement in method Clear_Disk of facility $DM$ in process Archiving contains a preemption point $pp_{a,3}$ due to the blocking device access. Therefore, according to the control flow, preemption point $pp_{a,1}$ at the start of the activity of process Archiving has two successor preemption points: $pp_{a,3}$ in the first execution of the body of the repetition in method Clear_Disk($ED$) and $pp_{a,2}$ at the end of the method call. With this notion in mind, a bead is now defined to be the unit of non-preemption.

**Definition 2.9 (Bead)** A bead is the non-preemttable part of an activity between a preemption point and all its immediate successors according to the branches defined in the program code.

A closer look at the types of preemption points reveals that a bead is either (1) a processing bead, when it represents the execution of a piece of code, or (2) a communication bead, when it represents part of a communication. Note that a single bead can not contain calls to methods of facilities
or objects of other processes. On the other hand, a single bead can contain method calls to facilities and objects of the same process. The activities and beads of the example control subsystem are depicted in Figure 2.7. Each box is a process, facility or object. Each straight line enclosed by small circles depicts a bead. A dashed arrow indicates a precedence relation between two successive beads. In addition, there is a precedence relation between each two successive beads that embrace a blocking device access. The figure also indicates the preemption point markers used in Figures 2.2 and 2.4. This is done to clarify how the beads are derived from the program code. The straight arrows tagged with the pair (4,6) indicates a relative timing constraint. This is discussed in detail in Section 2.4.2.

In the deterministic model, an activity is represented by a directed acyclic graph (DAG) of beads and precedence relations, that contains OR-branches and joins. Recall that preemption point \( pp_{a,3} \) has three successor preemption points \( pp_{a,4}, pp_{a,6} \) and \( pp_{a,8} \). Therefore, the part of the activity between the return of the delimited LPC and these three preemption points forms a single bead (see Figure 2.7). Recall that the while-statement in method Clear_Disk of facility DM in process Archiving contains a preemption point \( pp_{a,3} \) due to the blocking device access. Consequently, preemption point \( pp_{a,1} \) at the start of the activity of process Archiving has two successor preemption points, \( pp_{a,3} \) in the first execution of the body of the repetition and \( pp_{a,2} \) at the end of the method call. The second execution of the body of the while-statement has exactly the same structure. Note, however, that in this case preemption point \( pp_{a,3} \) has itself as successor preemption point. Thus, the repetition is represented pre-run-time by three beads (see Figure 2.7).

Summarizing, the dynamic structure of an application in the form of a set of activities is in the deterministic pre-run-time view represented by a set of beads \( B \). Since the activities are periodic and start at relative time 0, the dynamic structure only needs to be represented for the scheduling interval \([0,LCM]\), where LCM is the least common multiple of all the periods. If a pre-run-time schedule is constructed for the set of beads in this time interval, the same schedule can be used for successive time intervals of length LCM. Each bead \( b \in B \) executes within a process \( b, ps \in PS \). Additionally, each bead \( b \) belongs to an activity \( b, ac \) and has an execution time \( b, e \in \mathbb{N}^+ \). The execution time of an processing bead is the worst case execution time of the code. This can be calculated based on the execution times of the statements. State saving overhead due to a delimited LPC or an RPC is incorporated in the execution time of the bead in which the call is invoked. State rebuilding overhead due to a delimited LPC or an RPC is incorporated.
Figure 2.7: The activities and events of the control subsystem of a particle accelerator.
in the execution time of the first bead after the call. The execution time of a communication bead for a communication of data of size \( ds \) on a medium \( cm \) is delay \( \delta_{cm,ds} \). Each bead \( b \) that receives parameter data for a call or a return has a data size attribute \( b.ds \in \mathbb{N} \) that indicates the size of the data. The execution times of the communication beads that represent the passing of parameter data to bead \( b \) are calculated from this attribute. Finally, each communication bead \( b \) has an communication medium attribute \( b.cm \) that indicates the communication medium that is accessed during the bead.

The probability that a bead is selected to be executed at run-time depends on data-dependency in conditionals and repetitions of the application. Therefore, each bead \( b \in B \) has a so-called data-dependency probability \( b.ddp \) that indicates the probability that the bead is actually started at run-time. The data-dependency probability of a bead depends on the selection probabilities of conditionals and repetitions which it is part of. For instance, in the example control system, the bead between preemption point \( pp_{a,4} \) and preemption point \( pp_{d,1} \) is part of the then-clause of the if-then-else statement in method \textit{Sense.Particle}. Therefore, the data-dependency probability of this bead equals \( \frac{1}{15} \), because the then-clause has a selection probability of \( \frac{1}{15} \). In addition, the bead between preemption point \( pp_{a,6} \) and preemption point \( pp_{d,1} \) is part of the then-clause nested in the else-clause of the if-then-else statement in method \textit{Sense.Particle}. Therefore, the data-dependency probability of this bead equals \( \frac{7}{15} \times \frac{1}{15} = \frac{1}{25} \), because the else-clause has a selection probability of \( \frac{7}{15} \) and the nested then-clause has a selection probability of \( \frac{1}{25} \).

### 2.4 Scheduling Constraints on Beads

In addition to a set of beads that model the control flow of the activities of an application, the pre-run-time view contains several kinds of constraints on the execution of the beads. These constraints restrict the scheduling of beads. The scheduling constraints are formally expressed using a start time \( b.st \in \mathbb{N} \) for each bead \( b \in B \). Note that these start times have to be assigned during pre-run-time scheduling. As will become clear, the dynamic structure of an application together with the constraints necessary for pre-run-time scheduling is represented in the deterministic pre-run-time view by a triple \( (B, RTC, CC) \). The beads in \( B \) have absolute timing constraints. Set \( RTC \) is a set of relative timing constraints, including precedence relations and set \( CC \) is a set of consistency constraints. As discussed earlier, different
alternatives of the same conditional might be represented by different beads. Beads of different alternatives of the same conditional can be scheduled with overlap in time. However, this is not considered in the scheduling constraints and algorithms presented in this thesis.

2.4.1 Absolute Timing Constraints

For the sake of generality, the periods and deadlines of each activity are translated into an earliest start time \( b_{est} \) and a latest finishing time or deadline \( b.d \) per bead \( b \). Note that each bead belongs to a particular instance of a periodic activity. The start time of the first instance is 0, the start time of the second instance equals the period, the start time of the third instance equals twice the period and so on. The end time of an instance equals the start time of the next instance. Now, the earliest start time of a bead equals the start time of the instance to which the bead belongs. If the activity does not have an optional deadline, the deadline of the bead is the end time of the instance to which it belongs. Otherwise, the deadline of the bead is the start time of the instance plus the optional deadline. Earliest start times and deadlines are called absolute timing constraints, because they restrict the scheduling of a bead in a time interval absolute to the start time 0. Obviously, the absolute timing constraints have to be such that each bead can be scheduled between its earliest start time and its deadline

\[
(\forall b \in B : b_{est} + b.e \leq b.d).
\]

Absolute timing constraints require that the start time of a bead is such that the bead is completely executed between its earliest start time and its deadline

\[
(\forall b \in B : b_{est} \leq b_{st} \land b_{st} + b.e \leq b.d).
\]

2.4.2 Relative Timing Constraints

The designer can also enforce synchronization of activities by imposing relative timing constraints between beads. Relative timing constraints are necessary to synchronize different parts of an application. For example, when a robot repeatedly has to take a bottle from an assembly line, the part of the application that controls the robot arm has to synchronize with the part of the application that controls the assembly line. More formally, a relative
timing constraint \((so, lo)\) from a bead \(b_1\) to a bead \(b_2\) requires that the difference \(b_2.st - b_1.st\) between the start times of the beads is bounded from below by a smallest offset \(so \in \mathbb{N}\) and from above by a largest offset \(lo \in \mathbb{N}\)

\[
b_1.st + so \leq b_2.st \leq b_1.st + lo.
\]

Consequently, a relative timing constraint is a directed relation that constrains the scheduling of two beads relative to each other. Since the end time of a bead is equal to the start time of the bead plus its execution time, a relative timing constraint can also be used to implicitly relate the end times of two beads. Two beads can be forced to execute strictly parallel when a relative timing constraint \((0,0)\) is imposed on the start times of the beads.

For reasons of simplicity, the designer is allowed to add relative timing constraints between beads once the beads of the application have been derived from the processes, facilities and objects. In the control subsystem, the designer has imposed a relative timing constraint \((4,6)\) from the bead that starts at method \(DM\).Read.Disk\((d, report)\) with \(d\) equal to \(FD\) in the activity of process Reporting to the bead that starts at method \(DM\).Clear.Disk\((FD)\) in the activity of process Archiving (see Figure 2.7). This enforces the clearing of disk \(FD\) to take place between 4 and 6 time units after the reading of disk \(FD\). A similar relative timing constraint is imposed to enforce this for disk \(PD\).

For the sake of simplicity and generality, all kinds of precedence relations between beads of the same activity are represented by relative timing constraints. First, each normal precedence relation from a bead \(b_1\) to its successor \(b_2\) is a special case of a relative timing constraint and can be represented by \((b_1, e, \infty)\). Second, the precedence relation between two successive beads \(b_1\) and \(b_2\) that embrace a blocking access to device \(dr\), is represented by a relative timing constraint \((b_1.e + \alpha_{dr,ds}, \infty)\) from \(b_1\) to \(b_2\). Here, \(ds\) is the size of the data involved. Third, the precedence relation between two successive beads \(b_1\) and \(b_2\) of an asynchronous communication is a normal precedence relation, and is therefore represented by a relative timing constraint \((b_1.e, \infty)\) from \(b_1\) to \(b_2\). Finally, in a synchronous communication, the communication beads must be executed in parallel. Therefore, this is represented by a relative timing constraint \((0,0)\) on the start times of both beads. The set of relative timing constraints \(RTC\) contains a quadruple \((b_1, so, lo, b_2)\) for each relative timing constraint \((so, lo)\) from \(b_1\) to \(b_2\) that has to be satisfied. This set defines a directed acyclic graph on the set of beads \(B\), called the execution DAG. For convenience, a path in the execution
DAG is defined. A path \( b_1 \rightarrow b_n \) from bead \( b_1 \) to bead \( b_n \) is a sequence of \( n \) beads connected by precedence relations and relative timing constraints.

### 2.4.3 Consistency Constraints

Consistency of devices and communication media is only guaranteed if no two different accesses overlap in time unless at most one of the accesses will actually be executed at run-time due to data-dependency. This means that if two beads in different activities access the same medium or device, they have to be ordered in time. For a device, this additionally means that during the second phase of a blocking device access, no beads of other activities that access the same device may be scheduled.

All consistency requirements for devices and communication media are translated into a set of consistency constraints that demand a temporal ordering among subgraphs of beads. A subgraph of beads is only defined for beads of the same activity that are connected by precedence relations. A subgraph is denoted \((b_i \leftrightarrow b_j)\), where \( b_i \) is the bead that starts the subgraph and \( b_j \) is the bead that ends the subgraph. A consistency constraint between a subgraph of beads \((b_1 \leftrightarrow b_2)\) and a subgraph \((b_2 \leftrightarrow b_4)\) requires that \( b_1 \) starts after \( b_4 \) has finished or \( b_3 \) starts after \( b_2 \) has finished.

\[
b_4.st + b_4.e \leq b_1.st \lor b_2.st + b_2.e \leq b_3.st.
\]

The set of consistency constraints \( CC \) contains a quadruple \((b_1, b_2, b_3, b_4)\) for each consistency constraint between subgraphs \((b_1 \leftrightarrow b_2)\) and \((b_2 \leftrightarrow b_4)\) that has to be satisfied.

### 2.5 Assignment and Scheduling Problems

Based on the model developed in this chapter, the assignment and pre-run-time scheduling problems can now be formulated. At first, two problems are identified: the assignment of processes to processors and the scheduling of beads in time.

#### 2.5.1 Process Assignment Problem

Before the system can start executing an application, each process in \( PS \) has to be assigned to a processor in \( PR \). A process assignment is formally defined as follows.
Definition 2.10 (Process assignment) Let $PS$ be the set of processes and $PR$ the set of processors. Then, a process assignment $PA : PS \rightarrow PR \cup \{0\}$ is a function that maps each process in $PS$ to a processor in $PR$ or to 0. Thus, if $PA(ps) \in PR$, process $ps \in PS$ is assigned to processor $PA(ps)$. If $PA(ps) = 0$, process $ps$ is unassigned.

Note that a process assignment can be a partial process assignment in which there are unassigned processes. Therefore, the set of assigned processes based on a certain process assignment is defined as follows.

Definition 2.11 (Assigned process set) Let $PS$ be the set of processes and $PR$ the set of processors. Let $PA$ be a process assignment. Then, the set of assigned processes is defined as

$$APS(PA) = \{ ps \in PS \mid PA(ps) \in PR \}.$$ 

For convenience, a definition is given for the set of processes assigned to a particular processor based on a certain process assignment.

Definition 2.12 (Assigned process to processor set) Let $PS$ be the set of processes and $PR$ the set of processors. Let $PA$ be a process assignment and let $pr \in PR$. Then, the set of assigned processes to a processor is defined as

$$APS(pr, PA) = \{ ps \in PS \mid PA(ps) = pr \}.$$ 

A process assignment is correct if and only if the total number of devices implicitly assigned to each processor $pr \in PR$ does not exceed the maximum $pr.md$. This is formally expressed using a property that describes whether a processor is correct.

Property 2.1 Let $PR$ be the set of processors and let $pr \in PR$. Let $PA$ be a process assignment. Then, the correct processor property for $pr$ and $PA$ is defined as

$$CorrectProcessor(pr, PA) \iff \sum_{ps \in APS(pr, PA)} ps.md \leq pr.md.$$ 

□
2.5. Assignment and Scheduling Problems

The correct process assignment property is defined based on the correct processor property.

**Property 2.2** Let \( PR \) be the set of processors. Let \( PA \) be a process assignment. Then, the correct process assignment property for \( PA \) is defined as

\[
\text{CorrectProcessAssignment}(PA) \Rightarrow (\forall pr \in PR : \text{CorrectProcessor}(pr, PA)).
\]

Now, the process assignment problem is formally defined as follows.

**Problem 2.1 (Process assignment problem)** Let \( PS \) be the set of processes and \( PR \) the set of processors. Then, the process assignment problem is to find a correct process assignment \( PA \) in which each process in \( PS \) is assigned to a processor in \( PR \). Thus, \( \text{CorrectProcessAssignment}(PA) \) and \( APS(PA) = PS \) must hold.

The process assignment problem is equivalent to the bin packing problem with a fixed number of bins. The items to be packed are the processes, the size of each process is the number of devices managed by the process, and each processor is a bin. The bin packing problem with a fixed number of bins is known to be NP-complete [16]. Therefore, the process assignment problem is also NP-complete.

When a process is assigned to a processor, each bead that executes in the process is assigned to the same processor. The attribute \( b.pr \in PR \) indicates the processor to which bead \( b \in B \) is assigned. For convenience, a definition is given for the set of beads assigned to a particular processor.

**Definition 2.13** (Assigned bead to processor set) Let \( PR \) be the set of processors and let \( pr \in PR \). Let \( B \) be the set of beads. Then, the set of assigned beads to a processor is defined as

\[
AB(pr) = \{ b \in B \mid b.pr = pr \}.
\]

Once all beads are assigned, it is known which method calls require communication and which routes are used to pass parameter data. Thus, appropriate
communication beads that represent this parameter passing can be inserted and constraints can be adjusted. This is done according to the communication mechanisms described in Section 2.3.1. An example is given below. Inserted communication beads are added to the set of beads $B$ and the set of relative timing constraints $RTC$ is adjusted accordingly. In addition, consistency constraints between communication beads of different activities that use the same communication medium are added to the set of consistency constraints $CC$.

For example, suppose that two successive beads $b_i$ and $b_j$ of the same activity execute at processor $pr_i$ and processor $pr_j$ respectively. At the end of $b_i$ a call or a return from a call occurs and parameter data has to be passed from processor $pr_i$ to processor $pr_j$ via route $ro_{ij}$. The size of the parameter data to be passed equals $b_j.ds$. Suppose route $ro_{ij}$ consists of two hops: the first one from $pr_i$ to processor $pr_k$ via synchronous communication medium $cm_i$, and the second one from $pr_k$ to $pr_j$ via asynchronous medium $cm_j$. Thus, parameter passing from $pr_i$ to $pr_j$ consists of two successive inter-processor communications.

The inter-processor communication from $pr_i$ to $pr_k$ via synchronous medium $cm_i$, is represented by two communication beads $b_i^1$ and $b_k^2$ that access $cm_i$ as depicted in Figure 2.5b. Both beads are not part of a particular process, but they belong to the same activity as $b_i$ and $b_j$. Bead $b_i^1$ is assigned to processor $pr_i$, and bead $b_j^2$ is assigned to processor $pr_k$. The execution times of both beads is set to the delay $\delta_{cm_i, b_j, ds}$ for communicating data of size $b_j.ds$ via medium $cm_i$ as defined at the end of Section 2.3.1. Since $cm_i$ is a synchronous medium, there is a relative timing constraint $(0,0)$ from $b_i^1$ to $b_j^2$.

The inter-processor communication from $pr_k$ to $pr_j$ via asynchronous medium $cm_j$, is represented by two communication beads $b_k^1$ and $b_j^2$ that access $cm_j$ as depicted in Figure 2.5a. Both beads are not part of a particular process, but they belong to the same activity as $b_i$ and $b_j$. Bead $b_k^1$ is assigned to processor $pr_k$, and bead $b_j^2$ is assigned to processor $pr_j$. The execution times of both beads is set to the delay $\delta_{cm_j, b_j, ds}$ for communicating data of size $b_j.ds$ via medium $cm_j$ as defined at the end of Section 2.3.1. Since $cm_j$ is an asynchronous medium, there is a relative timing constraint $(b_i^1, e, \infty)$ from $b_i^1$ to $b_j^2$.

Finally, these communication beads are inserted between $b_i$ and $b_j$ by adding precedence relations from $b_i$ to $b_i^1$, from $b_j^2$ to $b_j^1$ and from $b_j^2$ to $b_j$. 
2.5. Assignment and Scheduling Problems

2.5.2 Bead Scheduling Problem

Once the processes of the application are assigned and communication is inserted, all beads have to be scheduled in time. This means that for each bead in $B$, a start time has to be found. A bead schedule is formally defined as a function $BS$ that maps the beads in $B$ to the set of non-negative naturals $\mathbb{N}$.

**Definition 2.14 (Bead schedule)** Let $B$ be a set of beads. Then, a bead schedule $BS : B \rightarrow \mathbb{N}$ is a function that maps each bead in $B$ to a start time in $\mathbb{N}$.

For convenience, the bead attribute $b.st$ is also used to indicate the start time of bead $b$. In the remainder, $BS(b)$ and $b.st$ are used interchangeably and bead schedule is sometimes abbreviated to schedule.

The bead schedule has to be feasible, i.e., no two different beads at the same processor may overlap in time, and all constraints must be satisfied. This is formally expressed using the following four properties. The first property describes whether no two different beads at the same processor overlap in time given a particular bead schedule.

**Property 2.3** Let $B$ be a set of beads assigned to a set of processors $PR$. Let $BS$ be a schedule for the beads in $B$. Then, the no overlap property for $BS$, $B$ and $PR$ is defined as

$NoOverlap(BS, B, PR) \iff (\forall pr \in PR : (\forall b_1, b_2 \in AB(pr) : b_1 \neq b_2 : BS(b_1) + b_1.e \leq BS(b_2) \lor BS(b_2) + b_2.e \leq BS(b_1) ))$.

The second property describes whether all absolute timing constraints are satisfied given a particular bead schedule.

**Property 2.4** Let $B$ be a set of beads and $BS$ a schedule for the beads in $B$. Then, the absolute timing constraints satisfied property for $BS$ and $B$ is defined as

$AbsoluteTimingConstraintsSatisfied(BS, B) \iff (\forall b \in B : b.est \leq BS(b) \land BS(b) + b.e \leq b.d)$.
The third property describes whether all relative timing constraints are satisfied given a particular bead schedule.

**Property 2.5** Let $B$ be a set of beads and $BS$ a schedule for the beads in $B$. Let $RTC$ be a set of relative timing constraints on the beads in $B$. Then, the relative timing constraints satisfied property for $BS$, $B$ and $RTC$ is defined as

$$
\text{RelativeTimingConstraintsSatisfied}(BS, B, RTC) \iff \quad \left( \forall b_1, b_2 \in B, so, lo \in \mathbb{N} : (b_1, so, lo, b_2) \in RTC : BS(b_1) + so \leq BS(b_2) \land BS(b_2) \leq BS(b_1) + lo \right).
$$

The fourth property describes whether all consistency constraints are satisfied given a particular bead schedule.

**Property 2.6** Let $B$ be a set of beads and $BS$ a schedule for the beads in $B$. Let $CC$ be a set of consistency constraints on the beads in $B$. Then, the consistency constraints satisfied property for $BS$, $B$ and $CC$ is defined as

$$
\text{ConsistencyConstraintsSatisfied}(BS, B, CC) \iff \quad \left( \forall b_1, b_2, b_3, b_4 \in B : (b_1, b_2, b_3, b_4) \in CC : BS(b_4) + b_4.e \leq BS(b_1) \lor BS(b_4) + b_2.e \leq BS(b_3) \right).
$$

Now, a particular bead schedule is feasible if and only if it satisfies Property 2.3 to Property 2.6. This expressed by the following property.

**Property 2.7** Let $B$ be a set of beads assigned to a set of processors $PR$. Let $BS$ be a schedule for the beads in $B$. Let $RTC$ be a set of relative timing constraints on the beads in $B$. Let $CC$ be a set of consistency constraints
2.6 Assignment and Scheduling Approach

on the beads in $B$. Then, the feasible bead schedule property for $BS$, $B$, $PR$, $RTC$, and $CC$ is defined as

$$FeasibleBeadSchedule(BS, B, PR, RTC, CC)$$

$$\Leftrightarrow$$

$$NoOverlap(BS, B, PR) \wedge$$

$$AbsoluteTimingConstraintsSatisfied(BS, B) \wedge$$

$$RelativeTimingConstraintsSatisfied(BS, B, RTC) \wedge$$

$$ConsistencyConstraintsSatisfied(BS, B, CC).$$

Now, the bead scheduling problem is formally defined as follows.

**Problem 2.2 (Bead scheduling problem)** Let $B$ be the set of beads assigned to a set of processors $PR$. Let $RTC$ and $CC$ be sets of relative timing constraints and consistency constraints on the beads in $B$. Then, the bead scheduling problem is to find a feasible bead schedule $BS : B \rightarrow \mathbb{N}$, i.e., such that $FeasibleBeadSchedule(BS, B, PR, RTC, CC)$ holds.

The bead scheduling problem is an extension of the problem of finding a schedule for a set of non-preemptable tasks with release times and deadlines at a single processor. The latter problem is known to be NP-complete [15]. Therefore, the bead scheduling problem is also NP-complete.

2.6 Assignment and Scheduling Approach

Now that the assignment and pre-run-time scheduling problems are defined, an approach is described that can solve these problems (see Figure 2.8). The application designer makes an application that consists of processes, facilities and objects using a library of pre-defined ADTs and ADOs. The designer also describes the hardware architecture on which the application has to be executed. Subsequently, the designer models the application and the hardware architecture and passes this to the first step of the assignment and pre-run-time scheduling approach. The application is modeled as a set of processes $PS$, a set of beads $B$ with absolute timing constraints, a set of relative timing constraints $RTC$ and a set of consistency constraints $CC$. The hardware architecture is modeled as a set of processors $PR$, a set of communication media $CM$ and a set of routes $RO$. 

2.6.1 Process Assignment

In the first step of the approach, a correct process assignment $PA$ of the processes in $PS$ to the set of processors $PR$ is searched for [45]. This is done using the set of communication media $CM$, the set of routes $RO$, the set of beads $B$, the set of relative timing constraints $RTC$ and the set of consistency constraints $CC$. The process assignment problem is already discussed in detail in Section 2.5.1, where it is shown that the problem is NP-complete. However, although the number of devices assigned to a processor is obviously a constraining factor, it is assumed that it is relatively easy to find a process assignment that satisfies Property 2.2. If a correct process assignment is not found, the application must be redesigned. Otherwise, the set of beads and the sets of constraints are extended with communication beads and corresponding constraints as described in Section 2.5. Then, the process assignment $PA$, the new set of beads $B$ and the new sets of constraints $RTC$ and $CC$ are passed to the second step of the approach. In Chapter 3, an algorithm is presented that can solve the process assignment problem.
2.6. Assignment and Scheduling Approach

After assignment and communication insertion, a feasible bead schedule $BS$ must be found for the set of beads $B$ assigned to the set of processors $PR$ with constraint sets $RTC$ and $CC$. As discussed in the previous section, the problem of finding a feasible bead schedule is NP-complete. At this point, it should be emphasized that the main goal of the approach is to find a bead schedule that satisfies all constraints. In other words, there are no additional optimality criteria that a bead schedule has to satisfy, such as minimum total execution time [3]. Consequently, a bead schedule in which all beads are spread over the entire scheduling interval $[0, LCM)$ is as good as a bead schedule with minimum total execution time, as long as all the constraints are satisfied. Experiments with a first prototype scheduler [42] showed that scheduling a particular bead can have an enormous influence on the scheduling of beads on other processors due to precedence relations. These two facts have lead to the idea to perform bead scheduling in two stages, respectively the second and third step of the approach.

In the second step of the approach, a time interval, called *window*, is assigned to each bead in $B$. This is done such that all constraints are satisfied when each bead is scheduled in its window. If windows can not be found, the application must be redesigned. Otherwise, the set of beads $B$ with windows is passed to the third step of the approach.

In the third step, a start time is assigned to each bead. This is done such that each bead is scheduled in its window and no two beads at the same processor overlap in time. If start times are found this way, a feasible bead schedule is found. Then, the correct process assignment and feasible bead schedule can be used to execute the application. Otherwise, the application must be redesigned.

Note that constraints among beads at different processors are incorporated before the beads are actually scheduled. Thus, in the third step, the scheduling of a bead at a particular processor has no influence on the scheduling of beads at other processors. In addition, scheduling the beads at each processor can be solved in isolation. Furthermore, the windows can be assigned such that the beads are spread over the entire scheduling interval. These are the advantages of the window assignment approach. Also, hopefully the window approach will lead to only a small amount of backtracking. Performance measurements in Chapter 6 show that this is indeed the case for all tested applications.

A disadvantage of the approach is obviously that a window assignment can not be found, even if one exists. However, in this thesis, it is shown that in most cases a window assignment can be found using efficient heuristics
and conditions. In the following, the window assignment problem and the resulting scheduling problem are defined more formally. In addition, it is proven that the bead scheduling problem can be solved by solving the window assignment problem and subsequent scheduling problem.

2.6.2 Window Assignment

A window assignment is formally defined as a function $WA$ that maps the beads in $B$ to pairs of naturals $\mathbb{N} \times \mathbb{N}^+$.

**Definition 2.15 (Window assignment)** Let $B$ be a set of beads. Then, a window assignment $WA : B \rightarrow \mathbb{N} \times \mathbb{N}^+$ is a function that maps each bead in $B$ to a time interval in $\mathbb{N} \times \mathbb{N}^+$. For a bead $b \in B$, $WA(b).ws$ is the window start time and $WA(b).we$ is the window end time of the window of $b$.

For convenience, the bead attributes $b.ws$ and $b.we$ are also used to indicate the window start and end time of bead $b$. In the remainder, $WA(b).ws$ and $b.ws$ as well as $WA(b).we$ and $b.we$ are used interchangeably.

The window assignment has to be correct, i.e., each bead must fit in its window, and all constraints have to be satisfied when each bead is scheduled in its window. The following property describes whether each bead is scheduled in its window given a particular bead schedule.

**Property 2.8** Let $B$ be a set of beads and $WA$ a window assignment to the beads in $B$. Let $BS$ be a schedule for the beads in $B$. Then, the beads in windows property for $BS$, $WA$ and $B$ is defined as

$$\text{BeadsInWindows}(BS, WA, B)$$

$$\Leftrightarrow$$

$$(\forall b \in B : WA(b).ws \leq BS(b) \wedge BS(b) + b.c \leq WA(b).we).$$

Now, the correctness of a window assignment is formally expressed using the following four properties. The first property describes whether each bead fits in its window given a particular window assignment.

**Property 2.9** Let $B$ be a set of beads and $WA$ a window assignment to the beads in $B$. Then, the minimum window width property for $WA$ and $B$
2.6. Assignment and Scheduling Approach

is defined as

\[ \text{MinimumWindowWidth}(WA, B) \]
\[ \iff \]
\[ (\forall b \in B : \ WA(b).ws + b.e \leq \ WA(b).we). \]

\[ \square \]

The following lemma states that this property assures that each bead can be scheduled in its window. Thus, if a window assignment \( WA \) to the set of beads \( B \) satisfies Property 2.9, there exists a bead schedule \( BS \) that satisfies Property 2.8.

Lemma 2.1 Let \( B \) be a set of beads and \( WA \) a window assignment to the beads in \( B \). Then, the following holds:

\[ \text{MinimumWindowWidth}(WA, B) \]
\[ \implies \]
\[ (\exists BS : \text{BeadsInWindows}(BS, WA, B)). \]

Proof Consider a \( BS \) for which holds \((\forall b \in B : BS(b) = WA(b).ws)\). Hence, the right hand side of the implication can be simplified to \((\forall b \in B : \ WA(b).ws + b.e \leq \ WA(b).we)\) by substituting \( WA(b).ws \) for \( BS(b) \). \( \square \)

The second property assures that all absolute timing constraints are satisfied when each bead is scheduled in its window. Therefore, the window of each bead must be contained in the time interval between the earliest start time and the deadline of the bead.

Property 2.10 Let \( B \) be a set of beads and \( WA \) a window assignment to the beads in \( B \). Then, the absolute timing constraints assured property for \( WA \) and \( B \) is defined as

\[ \text{AbsoluteTimingConstraintsAssured}(WA, B) \]
\[ \iff \]
\[ (\forall b \in B : \ b.est \leq \ WA(b).ws \land \ WA(b).we \leq \ b.d). \]

\[ \square \]

The following lemma states that this property assures that all absolute timing constraints are satisfied when each bead is scheduled in its window. Thus, if a window assignment \( WA \) to the set of beads \( B \) satisfies Property 2.8, all bead schedules that satisfy Property 2.8 also satisfy Property 2.4.
Lemma 2.2 Let \( B \) be a set of beads and \( WA \) a window assignment to the beads in \( B \). Then, the following holds:

\[
\text{AbsoluteTimingConstraintsAssured}(WA, B) \Rightarrow (\forall BS: \text{BeadsInWindows}(BS, WA, B): \text{AbsoluteTimingConstraintsSatisfied}(BS, B)).
\]

**Proof** Suppose, \( WA \) is such that \((\forall b \in B: \text{b.est} \leq WA(b).ws \land WA(b).we \leq b.d)\) holds. Consider a bead schedule \( BS \) for which \((\forall b \in B: WA(b).ws \leq BS(b) \land BS(b) + b.e \leq WA(b).we)\). These two predicates combined imply \((\forall b \in B: \text{b.est} \leq BS(b) \land BS(b) + b.e \leq b.d)\).

The third property assures that all relative timing constraints are satisfied when each bead is scheduled in its window. Therefore, for each relative timing constraint \((so, lo)\) from a bead \( b_1 \) to a bead \( b_2 \), \(1\) the latest start time of \( b_1 \) plus the smallest offset must be at most the smallest start time of \( b_2 \); \(b_1.ws - b_1.e + so \leq b_2.ws\), and \(2\) the latest start time of \( b_2 \) must be at most the smallest start time of \( b_1 \) plus the largest offset \( lo; b_2.ws - b_2.e \leq b_1.ws + lo\).

**Property 2.11** Let \( B \) be a set of beads and \( RTC \) a set of relative timing constraints on the beads in \( B \). Let \( WA \) be a window assignment to the beads in \( B \). Then, the relative timing constraints assured property for \( WA, B \) and \( RTC \) is defined as

\[
\text{RelativeTimingConstraintsAssured}(WA, B, RTC) \Leftrightarrow (\forall b_1, b_2 \in B, so, lo \in \mathbb{N}: (b_1, so, lo, b_2) \in RTC : \begin{align*}
WA(b_1).ws - b_1.e + so & \leq WA(b_2).ws \land \\
WA(b_2).ws - b_2.e & \leq WA(b_1).ws + lo
\end{align*}).
\]

The following lemma states that this property assures that all relative timing constraints are satisfied when each bead is scheduled in its window. Thus, if a window assignment \( WA \) to the set of beads \( B \) with a set of relative timing constraints \( RTC \) satisfies Property 2.11, all bead schedules that satisfy Property 2.8 also satisfy Property 2.5.
2.6. Assignment and Scheduling Approach

Lemma 2.3 Let \( B \) be a set of beads and RTC a set of relative timing constraints on the beads in \( B \). Let \( WA \) be a window assignment to the beads in \( B \). Then, the following holds:

\[
RelativeTimingConstraintsAssured(WA, B, RTC) \Rightarrow (\forall BS: \text{BeadsInWindows}(BS, WA, B) : RelativeTimingConstraintsSatisfied(BS, B, RTC)).
\]

Proof Suppose, \( WA \) is such that \((\forall b_1, b_2 \in B, s_0, l_0 \in \mathbb{N} : (b_1, s_0, l_0, b_2) \in RTC : WA(b_1).we - b_1.e + s_0 \leq WA(b_2).ws \land WA(b_2).we - b_2.e \leq WA(b_1).ws + l_0)\). Consider a bead schedule \( BS \) for which \((\forall b \in B : WA(b).ws \leq BS(b) - BS(b) + b.e \leq WA(b).we)\). These two predicates combined imply \((\forall b_1, b_2 \in B, s_0, l_0 \in \mathbb{N} : (b_1, s_0, l_0, b_2) \in RTC : BS(b_1) + s_0 \leq BS(b_2) - BS(b_2) \leq BS(b_1) + l_0)\). The fourth property assures that all consistency constraints are satisfied when each bead is scheduled in its window. Therefore, for each consistency constraint \((b_1, b_2, b_3, b_4) \in CC\), either the window of \( b_4 \) must end before the window of \( b_1 \) starts or the window of \( b_2 \) must end before the window of \( b_3 \) starts.

Property 2.12 Let \( B \) be a set of beads and \( CC \) a set of consistency constraints on the beads in \( B \). Let \( WA \) be a window assignment to the beads in \( B \). Then, the consistency constraints assured property for \( WA, B \) and \( CC \) is defined as

\[
ConsistencyConstraintsAssured(WA, B, CC) \Rightarrow (\forall b_1, b_2, b_3, b_4 \in B : (b_1, b_2, b_3, b_4) \in CC : WA(b_4).we \leq WA(b_1).ws \lor WA(b_2).we \leq WA(b_3).ws).
\]

The following lemma states that this property assures that all consistency constraints are satisfied when each bead is scheduled in its window. Thus, if a window assignment \( WA \) to the set of beads \( B \) with a set of consistency constraints \( CC \) satisfies Property 2.12, all bead schedules that satisfy Property 2.8 also satisfy Property 2.6.
Lemma 2.4 Let $B$ be a set of beads and $CC$ a set of consistency constraints on the beads in $B$. Let $WA$ be a window assignment to the beads in $B$. Then, the following holds:

$$ConsistencyConstraintsAssured(WA, B, CC) \Rightarrow (\forall BS: \text{BeadsInWindows}(BS, WA, B): ConsistencyConstraintsSatisfied(BS, B, CC))$$

Proof Suppose, $WA$ is such that $(\forall b_1, b_2, b_3, b_4 \in B: (b_1, b_2, b_3, b_4) \in CC: WA(b_4).wc \leq WA(b_1).ws \lor WA(b_2).we \leq WA(b_3).ws)$. Consider a bead schedule $BS$ for which $(\forall b \in B: WA(b).ws \leq BS(b) \land BS(b) + b.e \leq WA(b).we)$. These two predicates combined imply $(\forall b_1, b_2, b_3, b_4 \in B: (b_1, b_2, b_3, b_4) \in CC: BS(b_4) + b_4.e \leq BS(b_1) \lor BS(b_2) + b_2.e \leq BS(b_3))$.

Now, a particular window assignment is correct if and only if it satisfies Property 2.9 to Property 2.12. This is expressed by the following property.

Property 2.13 Let $B$ be a set of beads and $WA$ a window assignment to the beads in $B$. Let $RTC$ be a set of relative timing constraints on the beads in $B$. Let $CC$ be a set of consistency constraints on the beads in $B$. Then, the correct window assignment property for $WA, B, RTC$ and $CC$ is defined as

$$CorrectWindowAssignment(WA, B, RTC, CC) \Leftrightarrow MinimumWindowWidth(WA, B) \land AbsoluteTimingConstraintsAssured(WA, B) \land RelativeTimingConstraintsAssured(WA, B, RTC) \land ConsistencyConstraintsAssured(WA, B, CC).$$

Now, suppose a correct window assignment is found. Then, Lemma 2.1 implies that each bead fits in its window. Suppose that a bead schedule is found in which each bead is scheduled in its window. Then, Lemma 2.2 to Lemma 2.4 imply that all constraints are satisfied. Thus, Property 2.4 to Property 2.6 are satisfied for the bead schedule. Now, according to Property
2.6. Assignment and Scheduling Approach

2.7, the bead schedule is feasible if and only if it also satisfies the no overlap Property 2.3. This is expressed by the following theorem.

**Theorem 2.1** Let $B$ be a set of beads assigned to a set of processors $PR$. Let $WA$ be a window assignment to the beads in $B$. Let $RTC$ be a set of relative timing constraints on the beads in $B$. Let $CC$ be a set of consistency constraints on the beads in $B$. Let $BS$ be a schedule for the beads in $B$. Then, the following holds:

\[
\text{CorrectWindowAssignment}(WA, B, RTC, CC) \land \text{BeadsInWindows}(BS, WA, B) \land \text{NoOverlap}(BS, B, PR) \Leftrightarrow \text{FeasibleBeadSchedule}(BS, B, PR, RTC, CC).
\]

**Proof** First, $\text{CorrectWindowAssignment}(WA, B, RTC, CC)$ implies that $\text{MinimumWindowWidth}(WA, B)$ holds. Then, Lemma 2.1 implies that there is a bead schedule $BS$ that satisfies $\text{BeadsInWindows}(BS, WA, B)$. Second, $\text{CorrectWindowAssignment}(WA, B, RTC, CC)$ implies that $\text{AbsoluteTimingConstraintsAssured}(WA, B)$ holds. Then, since $\text{BeadsInWindows}(BS, WA, B)$ holds, Lemma 2.2 implies that $\text{AbsoluteTimingConstraintsSatisfied}(BS, B)$ holds. Similar reasoning can be done to show that $\text{RelativeTimingConstraintsSatisfied}(BS, B, RTC)$ and $\text{ConsistencyConstraintsSatisfied}(BS, B, CC)$ hold using Lemmas 2.3 and 2.4. Finally, $\text{NoOverlap}(BS, B, PR)$ already holds. \qed

The first goal of the second step of the approach is to find a correct window assignment. However, it is desirable that a so-called feasible window assignment is found for which a feasible bead schedule exists. This problem is an extension of the decision problem whether a single processor schedule exists for a set of tasks with release times and deadlines. The latter problem is known to be NP-complete [16]. Hence, the problem of finding a feasible window assignment is also NP-complete. Fortunately, certain feasibility conditions can be used to guide a window assignment algorithm towards a feasible window assignment. A feasibility condition is a condition that must be satisfied by a window assignment in order to be feasible. In this thesis, two feasibility conditions are identified that can be evaluated by a procedure with a time-complexity that is polynomial in the number of beads. Obviously, it can not be guaranteed that a feasible window assignment is always found.
The first feasibility condition states that for each pair of beads at the same processor, the sum of the execution times of the beads must be at most the length of the maximum interval in which both beads have to be scheduled.

Property 2.14 Let $B$ be a set of beads assigned to a set of processors $PR$. Let $WA$ be a window assignment to the beads in $B$. Then, the feasible pairs property for $WA$, $B$ and $PR$ is defined as

$$FeasiblePairs(WA, B, PR)$$

$$\iff$$

$$\forall pr \in PR : (\forall b_1, b_2 \in AB(pr) : b_1 \neq b_2 :$$

$$b_1.e + b_2.e \leq WA(b_2).we - WA(b_1).ws \ \vee$$

$$b_1.e + b_2.e \leq WA(b_1).we - WA(b_2).ws$$

$$)$$

It can be proven that if $FeasiblePairs(WA, B, PR)$ does not hold, $BeadsInWindow(BS, WA, B)$ and $NoOverlap(BS, B, PR)$ cannot both be satisfied for each bead schedule $BS$. Then, Theorem 2.1 implies that a feasible bead schedule does not exist for the window assignment and the window assignment is not feasible. Consider therefore two beads $b_1$ and $b_2$ at the same processor and assume that $FeasiblePairs(WA, B, PR)$ does not hold for these beads. Thus, $b_1.e + b_2.e > WA(b_2).we - WA(b_1).ws$ and $b_1.e + b_2.e > WA(b_1).we - WA(b_2).ws$. Suppose that $BS$ is a bead schedule for which $NoOverlap(BS, B, PR)$ holds. In addition, suppose that $b_2$ is scheduled after $b_1$ in $BS$ and that $b_1$ is scheduled in its window. Then, Property 2.8 implies $WA(b_1).ws \leq BS(b_1)$. $NoOverlap(BS, B, PR)$ implies that $b_2$ starts after $b_1$ has finished. Thus, $WA(b_1).ws + b_1.e \leq BS(b_2)$ which is equivalent to $WA(b_1).ws + b_1.e + b_2.e \leq BS(b_2) + b_2.e$. Finally, the initial assumption implies that $BS(b_2) + b_2.e > WA(b_2).we$. Thus, $b_2$ is not scheduled in its window and Property 2.8 does not hold for $WA$. Similar reasoning can be done for the case where $b_1$ is scheduled after $b_2$.

The second feasibility condition ensures that it is possible to accommodate all beads that must execute between a particular window start time and a subsequent window end time:

Property 2.15 Let $B$ be a set of beads assigned to a set of processors $PR$. Let $WA$ be a window assignment to the beads in $B$. Then, the feasible sets
2.6. Assignment and Scheduling Approach

property for $WA$, $B$ and $PR$ is defined as

$$\text{FeasibleSets}(WA, B, PR)$$

$$\Leftrightarrow$$

$$\forall pr \in PR: (\forall b_1, b_2 \in AB(pr): WA(b_1).ws < WA(b_2).we :$$

$$\sum b_3 \in AB(pr): WA(b_1).ws \leq WA(b_3).ws \land$$

$$WA(b_3).we \leq WA(b_2).we : b_3.e)$$

$$\leq WA(b_2).we - WA(b_1).ws)$$

$$\Box$$

It can be proven that if $\text{FeasibleSets}(WA, B, PR)$ does not hold, $\text{BeadsInWindows}(BS, WA, B)$ and $\text{NoOverlap}(BS, B, PR)$ can not both be satisfied for each bead schedule $BS$. Then, Theorem 2.1 implies that a feasible bead schedule does not exist for the window assignment and the window assignment is not feasible. Consider therefore two beads $b_1$ and $b_2$ at the same processor $pr$ for which $WA(b_1).ws < WA(b_2).we$ holds, but $\text{FeasibleSets}(WA, B, PR)$ does not hold. Thus, $\sum b_3 \in AB(pr): WA(b_1).ws \leq WA(b_3).ws \land WA(b_3).we \leq WA(b_2).we : b_3.e) > WA(b_2).we - WA(b_1).ws$. Suppose that $BS$ is a bead schedule for which $\text{NoOverlap}(BS, B, PR)$ holds. In addition, suppose that $b_m$ is the bead that is scheduled first among all the beads that contribute to the sum. Furthermore, suppose that $b_n$ is the bead that is scheduled last among all the beads that contribute to the sum. Suppose that $b_m$ is scheduled in its window. Then, Property 2.8 implies $WA(b_m).ws \leq BS(b_m)$ and since $b_m$ contributes to the sum, $WA(b_1).ws \leq BS(b_m)$ holds. Then, $\text{NoOverlap}(BS, B, PR)$ implies that $b_n$ starts after all other beads that contribute to the sum, because these beads have been executed without overlap in time. Thus, $WA(b_1).ws + \sum b_3 \in AB(pr) \setminus \{b_n\}: WA(b_1).ws \leq WA(b_3).ws \land WA(b_3).we \leq WA(b_2).we : b_3.e) \leq BS(b_n)$. This is equivalent to $WA(b_1).ws + \sum b_3 \in AB(pr): WA(b_1).ws \leq WA(b_3).ws \land WA(b_3).we \leq WA(b_2).we : b_3.e) \leq BS(b_n) + b_n.e$. Now, the initial assumption implies that $BS(b_n) + b_n.e > WA(b_2).we$. Finally, since $b_n$ contributes to the sum, $BS(b_n) + b_n.e > WA(b_n).we$. Thus, $b_n$ is not scheduled in its window and Property 2.8 does not hold for $WA$.

Note that Property 2.15 does not imply Property 2.14. A correct window assignment that satisfies Property 2.14 and Property 2.15 is called a presumably feasible window assignment. Note that a presumably feasible window
assignment is correct and thus also satisfies Property 2.13.

**Property 2.16** Let $B$ be a set of beads assigned to a set of processors $PR$. Let $WA$ be a window assignment to the beads in $B$. Let $RTC$ be a set of relative timing constraints on the beads in $B$. Let $CC$ be a set of consistency constraints on the beads in $B$. Then, the presumably feasible window assignment property for $WA$, $B$, $PR$, $RTC$ and $CC$ is defined as

$$\text{PresumablyFeasibleWindowAssignment}(WA, B, PR, RTC, CC)$$

$$\iff$$

$$\text{CorrectWindowAssignment}(WA, B, RTC, CC) \land$$

$$\text{FeasiblePairs}(WA, B, PR) \land$$

$$\text{FeasibleSets}(WA, B, PR).$$

Now, the window assignment problem is formally defined as follows.

**Problem 2.3 (Window assignment problem)** Let $B$ be the set of beads assigned to a set of processors $PR$. Let $RTC$ and $CC$ be sets of relative timing constraints and consistency constraints on the beads in $B$. Then, the window assignment problem is to find a presumably feasible window assignment $WA : B \rightarrow \mathbb{N} \times \mathbb{N}^+$, i.e., such that $\text{PresumablyFeasibleWindowAssignment}(WA, B, PR, RTC, CC)$ holds.

The problem of finding a presumably feasible window assignment is probably NP-complete, because (1) the consistency constraints and the feasibility conditions require a temporal ordering among beads, (2) the number of orderings is exponential in the number of beads, and (3) it has not yet been proven that the maximum number of possible orderings that need to be considered is polynomial in the number of beads. In Chapter 4, an algorithm is presented that can solve the window assignment problem.

### 2.6.3 Local Scheduling

A presumably feasible window assignment $WA$ is passed to the third step of the approach, where a schedule $BS$ for the set of beads $B$ has to be found. According to Theorem 2.1, it suffices to find a bead schedule $BS$ for which $\text{BeadsInWindows}(BS, WA, B)$ and $\text{NoOverlap}(BS, B, PR)$ holds. Thus, the bead schedule has to be such that each bead is scheduled in its window and no two beads at the same processor overlap in time. As mentioned
earlier, this problem can be solved by scheduling the beads at each processor in isolation. Thus, in the third step of the approach, the beads at each processor are scheduled locally. In order to specify this local scheduling problem more formally, the definition of a local schedule for a set of beads $PB$ at the same processor is given first.

**Definition 2.16 (Local schedule)** Let $PB$ be the set of beads at the same processor. Then, a local schedule $LS(PB)$ for the set of beads $PB$ is a sequence $LS(PB) = \{b_1, ..., b_{|PB|}\}$ of the beads in $PB$. Thus $(\forall i : 1 \leq i \leq |PB| : b_i \in PB)$. For convenience, $st_i = b_i.st$, $e_i = b_i.e$, $ws_i = b_i.ws$ and $we_i = b_i.wc$. The start times of all the beads in $LS(PB)$ are defined as

$$st_1 = ws_1 \land (\forall i : 1 < i \leq |PB| : st_i = \text{Max}(ws_i, st_{i-1} + e_{i-1})).$$

A local schedule is defined to be feasible if each bead in the local schedule finishes before its window end time.

**Property 2.17** Let $(b_1, ..., b_{|PB|})$ be a local schedule for a set of beads $PB$. Then, the feasible local schedule property for $(b_1, ..., b_{|PB|})$ is defined as

$$\text{FeasibleLocalSchedule}((b_1, ..., b_{|PB|})) \iff (\forall i : 1 \leq i \leq |PB| : st_i + e_i \leq we_i).$$

When a feasible local schedule is found for each processor in $PR$, an overall bead schedule $BS$ is found that satisfies $\text{BeadsInWindows}(BS, WA, B)$ and $\text{NoOverlap}(BS, R, PR)$. This is expressed by the following theorem.

**Theorem 2.2** Let $B$ be a set of beads assigned to a set of processors $PR$. Let $WA$ be a correct window assignment to the beads in $B$. Let $LS(AB(pr))$ be a local schedule for the beads at processor $pr$. Let $BS$ be the bead schedule based on the local schedules for all the processors, i.e., $(\forall b \in B : BS(b) = b.st)$. Then, the following holds:

$$(\forall pr \in PR : \text{FeasibleLocalSchedule}(LS(AB(pr))))$$

$$\iff \text{BeadsInWindows}(BS, WA, B) \land \text{NoOverlap}(BS, B, PR).$$
Proof. In Definition 2.16, the start times of the beads in a local schedule are defined such that the beads do not overlap in time. Since BS is based on a local schedule for each processor in PR, NoOverlap(BS, B, PR) holds. Note that the validity of this predicate does not depend on whether the local schedules are feasible or not. According to Definition 2.16, the start time of a bead b in a local schedule is at least the window start time of the bead: b.ws ≤ b.st. In addition, according to Property 2.17, the end time of a bead b in a feasible local schedule is at most the window end time of the bead: b.st + b.e ≤ b.we. Since the local schedule for each processor is feasible, each bead in B is scheduled in its window and thus BeadsInWindows(BS, WA, B) holds.

Now, the problem to be solved in the third step of the approach is, given a correct window assignment, find a feasible local schedule for each processor in PR. If these local schedules are found, Theorem 2.2 implies that a bead schedule is found that satisfies BeadsInWindows(BS, WA, B) and NoOverlap(BS, B, PR). Finally, Theorem 2.1 implies that this bead schedule is a feasible bead schedule. The local scheduling problem is formally defined as follows.

**Problem 2.4 (Local scheduling problem)** Let PB be a set of beads with windows assigned to the same processor. Then, the local scheduling problem is to find a feasible local schedule LS(PB) for a set of beads PB at the same processor, i.e., such that FeasibleLocalSchedule(LS(PB)) holds.

The local scheduling problem is equivalent to the problem of finding a schedule for a set of non-preemptable tasks with release times and deadlines. The tasks to be scheduled are the beads, each window start time is a release time and each window end time is a deadline. Therefore, the local scheduling problem is NP-complete [15]. However, several efficient algorithms exist that can solve this problem in reasonable time for small sets of beads [5, 6, 8, 53]. In Chapter 5, another algorithm is presented that can solve the local scheduling problem in reasonable time for even larger sets of beads [46].

### 2.7 Conclusion

In this chapter, an assignment and pre-run-time scheduling model for object-based applications on distributed real-time systems is presented. The static
2.7. Conclusion

structure of an application is modeled as a set of processes that create facilities and objects that privately manage devices. The dynamic structure of the application consists of a set of semi-preemptable activities which is modeled as a set of heads with consistency and timing constraints. These constraints include ones that were not yet considered in the context of pre-run-time scheduling, such as relative timing constraints. An approach is presented for the assignment and pre-run-time scheduling problems. In the following chapters, the three steps of the approach are discussed in detail and algorithms that can solve the various problems are presented. Finally, results of performance measurements of the pre-run-time scheduling approach are presented in Chapter 6.
Chapter 3

Process Assignment

In this chapter, a process assignment algorithm is presented that can solve the process assignment problem of the first step of the scheduling approach. The process assignment algorithm is heuristic and constructive and assigns processes in order of decreasing execution time. For this purpose, the algorithm constructs a set of candidate processors using several feasibility conditions. Then, a candidate processor is selected based on a novel objective function that weights the amount of communication against the amount of parallelism. If there is no candidate processor, the algorithm reassigns a process on the least full processor to a new processor. Results of performance measurements of the algorithm are presented in Chapter 6.

3.1 The Process Assignment Problem

The process assignment problem concerns a set of processes $PS$ and a set of beads $B$ that execute in the processes. The execution of the beads is restricted by (1) absolute timing constraints, (2) a set of relative timing constraints $RTC$ and (3) a set of consistency constraints $CC$. The processes have to be assigned to a set of processors $PR$ connected by a set of communication media $CM$. A set of routes $RO$ defines which media are used for communication between processors. As defined in Section 2.5, the process assignment problem is to find a correct process assignment $PA$ in which each process in $PS$ is assigned to a processor in $PR$. Thus,
CorrectProcessAssignment(PA) and APS(PA) = PS must hold. As discussed in Section 2.5, this problem is NP-complete.

Obviously, a process assignment for which there is a feasible bead schedule is preferred. The problem of finding such a process assignment is an extension of the decision problem whether there is a feasible bead schedule given a particular process assignment. This decision problem is known to be NP-complete [16]. Therefore, the problem of finding a process assignment for which there is a feasible bead schedule is also NP-complete. No literature has been found which treats this problem.

In this chapter, a constructive heuristic algorithm is presented that can find a correct process assignment. The algorithm is guided by a novel objective function and feasibility conditions. The objective function is based on several system properties defined in terms of the model presented in Chapter 2. Therefore, the problem is analyzed first.

3.2 Problem Analysis

Since a process is the unit of assignment to processors, several beads execute at each processor. Therefore, system properties of processes and processors are defined in terms of sets of beads.

The absolute timing constraints on a bead restrict the execution of the bead to a particular time interval. In addition, relative timing and consistency constraints may restrict the execution of the bead to an even smaller time interval. The entire time interval in which the bead can execute is called the execution interval of the bead. A probabilistic definition of the expected utilization for a set of beads that execute on the same processor is then defined using the execution intervals of the beads. Based on this definition, the expected amount of parallelism between two sets of beads that execute at two different processors is defined. Finally, the exact amount of communication via a medium is determined. For some of these properties, abstract procedures for their calculation are presented\(^1\).

3.2.1 Bead Execution Intervals

In a real-time system, tasks or activities usually can only be started after a certain time and have to be finished before a certain deadline. The formu-\(^1\)The abstract procedures are given without regard to implementation efficiency, only the functionality is expressed
tion of system properties becomes quite different from those of a traditional system in such a real-time setting. In particular, for each bead an execution interval can be defined in which the bead has to be executed. The execution interval of a bead \( b \in B \) is represented by an execution interval start time \( b.\text{eis} \) and an execution interval end time \( b.eie \). Obviously, the earliest start time and deadline of a bead define a lower and upper bound on the execution interval. Therefore, the execution intervals of all the beads in \( B \) have to satisfy the following execution interval constraint:

\[
(\forall \ b \in B : \ b.\text{eas} \leq b.\text{eis} \wedge b.\text{eie} \leq b.d).
\] (3.1)

Note that this constraint is equivalent to Property 2.10 that restricts the windows of the beads.

Each relative timing constraint \((so,lo)\) from a bead \( b_1 \) to a bead \( b_2 \) imposes an additional constraint on the execution intervals of both beads. It is simply required that the relative timing constraint is satisfied for the smallest possible start times of both beads and for the largest possible start times of both beads. Therefore, the execution intervals of the beads in \( B \) have to satisfy the following execution interval constraint:

\[
(\forall \ b_1, b_2 \in B, so, lo \in \mathbb{N} : (b_1, so, lo, b_2) \in RTC : \\
b_1.\text{eis} + so \leq b_2.\text{eis} \leq b_1.\text{eis} + lo \wedge \\
b_1.\text{eie} - b_1.e + so \leq b_2.\text{eie} - b_2.e \leq b_1.\text{eie} - b_1.e + lo 
). \] (3.2)

Note that this constraint resembles but is weaker than Property 2.11.

Suppose that the execution interval of a bead in \( B \) is such that constraints (3.1) and (3.2) are not satisfied. Then, it can easily be verified that when the bead executes in its execution interval an absolute or relative timing constraint on the bead might not be satisfied. Consequently, (3.1) and (3.2) are necessary constraints on the execution intervals. In addition, if the execution interval start time of a bead is as small as possible and the execution interval end time of the bead is as large as possible, the execution interval of the bead is the largest interval in which the bead can execute.

In Figure 3.1, a procedure is presented that adjusts the execution intervals of a set of beads \( B \) with a set of relative timing constraints \( RTC \) according to constraint (3.2). It is assumed that the execution intervals already satisfy constraint (3.1). The procedure repeatedly passes forwards and backwards through the execution DAG of beads, while execution interval start times are increased and execution interval end times are decreased
Figure 3.1: The procedure that adjusts the execution intervals of a set of beads $B$ to a set of relative timing constraints $RTC$. 

```
Adjust_Execution_Intervals_To_Relative_Timing_Constraints (in: $H$, $RTC$; out: FOUND);
FOUND := true; changed := true;
while changed do
  changed := false;
  /* adjust execution intervals in a forward direction */
  $R := \emptyset$; /* $R$ is the set of ready beads */
  $C := \{ b \in B \mid \exists b', so, lo : (b', so, lo, b) \in RTC \}$;
  /* $C$ is the set of candidate beads */
  while $C \neq \emptyset$ do
    $b_1 := \text{a random element from } C$;
    $C \setminus \{b_1\}; R \cup \{b_1\}$;
    for each $b_2 \in B : (b_1, so, lo, b_2) \in RTC$ do
      if $b_2.\text{ets} < b_1.\text{ets} + so$ then $b_2.\text{ets} := b_2.\text{ets} + so$; changed := true;
      if $b_2.\text{ere} > b_1.\text{ere} - b_2.\text{e} + b_2.\text{e} + lo$ then $b_2.\text{ere} := b_2.\text{ere} - b_1.\text{e} + b_2.\text{e} + lo$; changed := true;
      if $b_2.\text{ets} + b_2.\text{e} > b_2.\text{ere}$ then $\text{FOUND} := \text{false}; \text{Exit}$;
      /* $b_2$ becomes a candidate when all its predecessors are ready or candidate */
      if $(\forall b' : (b', so', lo', b) \in RTC : b' \in R \cup C)$ then $C \cup \{b_2\}$;
    end for;
  end while;
  /* adjust execution intervals in a backward direction */
  $R := \emptyset$; $C := \{ b \in B \mid \exists b', so, lo : (b, so, lo, b') \in RTC \}$;
  while $C \neq \emptyset$ do
    $b_2 := \text{Random.Element}(C)$;
    $C \setminus \{b_2\}; R \cup \{b_2\}$;
    for each $b_1 \in B : (b_1, so, lo, b_2) \in RTC$ do
      if $b_1.\text{ets} < b_2.\text{ets} - lo$ then $b_1.\text{ets} := b_2.\text{ets} - lo$; changed := true;
      if $b_1.\text{ere} > b_2.\text{ere} - b_2.\text{e} - so + b_1.\text{e}$ then $b_1.\text{ere} := b_1.\text{ere} - b_2.\text{e} - so - b_1.\text{e}$; changed := true;
      if $b_1.\text{ets} + b_1.\text{e} > b_1.\text{ere}$ then $\text{FOUND} := \text{false}; \text{Exit}$;
      /* $b_1$ becomes a candidate when all its successors are ready or candidate */
      if $(\forall b' : (b, so', lo', b') \in RTC : b' \in R \cup C)$ then $C \cup \{b_1\}$;
    end for;
  end while;
End while;
```

3.2. Problem Analysis

to satisfy relation (3.2). The start and end times are modified as little as possible. The outermost loop terminates when no execution interval start or end time was changed during an iteration, i.e., when constraint (3.2) is satisfied. Since execution interval start times are only increased and execution interval end times only decreased, it is possible that the width of the execution interval of a bead becomes smaller than the execution time of the bead. In this case, the absolute and relative timing constraints cannot be satisfied, because the execution interval is the largest possible execution interval of the bead. Therefore, the procedure terminates.

In the execution interval adjustment procedure, a forward pass through the execution DAG followed by a backward pass is called a pass of the procedure. During a pass, constraint (3.2) is checked for each relative timing constraint. Therefore, a pass has a time-complexity $O(|RTC|)$. If execution intervals that satisfy constraints (3.1) and (3.2) can be found, the number of relative timing constraints for which (3.2) is satisfied increases with at least one during each pass. Thus, the procedure can be terminated after a pass in which this number has not increased. Consequently, the number of passes is at most equal to $|RTC|$ and the procedure has a time-complexity $O(|RTC|)$.

Besides absolute and relative timing constraints, certain consistency constraints can be incorporated in the calculation of the execution intervals. To this end, each consistency constraint for which a temporal order is already determined is transformed into a precedence relation and the execution intervals are adjusted accordingly. It might often occur that a temporal order is already determined for a consistency constraint. Consider therefore two activities with the same period. Then, the execution interval of beads in different instances of the period have no overlap in time. Thus, a temporal order is already determined for a consistency constraint between such beads.

Now, consider a consistency constraint between a subgraph of beads $(b_1 \rightarrow b_2)$ and a subgraph of beads $(b_3 \rightarrow b_4)$. Obviously, the temporal order of the consistency constraint is already determined if there is a path in the execution DAG from a bead in one subgraph to a bead in the other subgraph. Additionally, the temporal order of the consistency constraint is also determined if: (1) $b_1$ cannot be finished before the latest start time of $b_2$: $b_1.e - b_1.c < b_2.e + b_2.c$, or (2) $b_2$ cannot be finished before the latest start time of $b_3$: $b_3.e - b_3.c < b_3.e + b_2.c$. In the first case, the subgraph of beads $(b_1 \rightarrow b_2)$ has to be executed before the subgraph of beads $(b_3 \rightarrow b_4)$ and a precedence relation $(b_2, \infty)$ from $b_2$ to $b_3$ is added to $RTC$. In the second case, the subgraph of beads $(b_3 \rightarrow b_4)$ has to be executed before the
subgraph of beads \((b_1 \xRightarrow{\phi} b_2)\) and a precedence relation \((b_4,e,\infty)\) from \(b_4\) to \(b_1\) is added to \(RTC\). In these cases, no feasible start times are discarded and the execution intervals remain the largest execution intervals of the beads.

The complete procedure for the calculation of execution intervals is shown in Figure 3.2. The execution interval of each bead is initialized to the interval between the earliest start time and the deadline of the bead. Thus, constraint (3.1) is satisfied. Subsequently, the execution intervals are adjusted to satisfy constraint (3.2) using the procedure of Figure 3.1. This initialization part of the procedure has a time-complexity \(O(|B| + |RTC|^2)\).

\begin{figure}[h]
\centering
\texttt{Calculate\_Execution\_Intervals}(\texttt{in:B,RTC,CC; out:FOUND});
\end{figure}

\begin{figure}[h]
\centering
\texttt{Adjust\_Execution\_Intervals\_To\_Relative\_Timing\_Constraints}(B,RTC,FOUND);
\end{figure}

\begin{figure}[h]
\centering
\texttt{Exit;}
\end{figure}
3.2. Problem Analysis

Then, consistency constraints for which a temporal order is already determined are transformed to precedence relations and the execution intervals are adjusted again to satisfy relation (3.2). Checking whether there is a path from one subgraph to another subgraph can be done by a standard procedure that passes through the execution DAG once. Therefore, the part of the procedure that checks whether a temporal order is already determined has a time-complexity \( O(|RTC|) \). Consistency constraint transformation continues until there are no more consistency constraints for which a temporal order is determined. Obviously, as soon as an execution interval becomes too small, the procedure terminates. However, the execution interval adjustment procedure of Figure 3.1 is called at most once for each consistency constraint. Therefore, the transformation part of the procedure has a time-complexity \( O(|RTC|^3 \times |CC|) \). The entire execution interval calculation procedure has a time-complexity \( O(|B| + |RTC|^2 \times |CC|) \). In the following, execution intervals are used to define several other system properties.

3.2.2 Expected Utilization

The next system property to be discussed is the utilization of a set of beads that execute on a particular processor. No literature has been found in which the utilization is defined for real-time systems. The real utilization of a processor in a particular time interval equals 1 when exactly one of the beads at the processor executes and 0 otherwise. Thus, the real utilization for a set of beads and can only be calculated when a feasible bead schedule has been constructed. However, as discussed in Section 2.5, finding a feasible bead schedule is NP-complete. Therefore, finding the real utilization is also an NP-complete problem. In general, the real utilization can only be estimated. An expected utilization is therefore defined which is based on the bead execution intervals and can be evaluated by a procedure with a polynomial time-complexity. In the remainder of this section, it is assumed that execution intervals for the set of beads \( B \) with sets of constraints \( RTC \) and \( CC \) are already calculated using the procedure of Figure 3.2.

The execution interval of a bead is used to define the probability that the bead executes. The probability that a bead \( b \) executes at time \( t \) is equal to the fraction of feasible bead schedules in which \( b \) executes at \( t \). Therefore, finding the execution probability of a bead is an NP-complete problem. Therefore, the expected execution probability is defined. Note that the data-dependency probability \( b.ddp \), introduced at the end of Section 2.3.3, is the probability that bead \( b \) is selected. The expected probability
that a bead executes depends on its data-dependency probability and on the
execution time of the bead and is equally divided over the execution interval
of the bead.

**Definition 3.1 (Expected execution probability)** Let $B$ be the set of
beads and let $b \in B$. Let $t \in \mathbb{N}$. Then, the expected execution probability
is defined as

$$
c_{e}(b, t) = \begin{cases} 
  b.\text{ddp} \times \frac{b.e/(b.eie - b.eis)}{b.eie} & \text{if } b.eis \leq t < b.eie \\
  0 & \text{otherwise.}
\end{cases}
$$

Note that the expected execution probability may exceed 1.

In the remainder, a set of beads $PB \subseteq B$ is considered that execute at
a particular processor. Then, the expected utilization for $PB$ at time $t$ can
be expressed as the likelihood that any of the beads in $PB$ executes at time
$t$.

**Definition 3.2 (Expected utilization)** Let $PB$ be a set of beads that
execute at a particular processor. Let $t \in \mathbb{N}$. Then, the expected utilization
is defined as

$$
e_{u}(PB, t) = \sum_{b \in PB} c_{e}(b, t).
$$

Since this definition does not define the real utilization, the expected utili-
zation may exceed 1. However, the expected utilization gives an indication
of the amount of contention between the beads in $PB$. In Figure 3.3, a
procedure is presented that calculates the expected utilization of a set of
beads $PB$ that execute at a particular processor. It is assumed that execu-
tion intervals are already calculated for the beads in $B$. The procedure first
calculates the expected execution probability of each bead in $PB$ according
to Definition 3.1. Then, the expected utilization is calculated according to
Definition 3.2. Note that the procedure in Figure 3.3 is defined for discrete
times $t$. However, the expected execution probability of a bead is constant
in the execution interval of the bead. In addition, the expected utilization
only changes at the start and end time of the execution interval of a bead.
Therefore, the procedure that calculates the expected utilization has a time-
complexity $O(|B|)$. 
Figure 3.3: The procedure for calculating the expected utilization for a set of beads $PB$ that execute at a particular processor.

3.2.3 Expected Parallelism

The expected utilization can be used to define other system properties such as the expected amount of parallelism between two sets of beads that execute at different processors. No literature has been found in which this property is defined in a real-time setting. Similar to the utilization, the real amount of parallelism depends on a particular bead schedule. Thus, the problem to determine this real amount is an NP-complete problem and the amount of parallelism can only be estimated. The expected amount of parallelism between two sets of beads $PB_1$ and $PB_2$ is the amount of time any bead in $PB_1$ executes simultaneously with any bead in $PB_2$:

\[ eapb(PB_1, PB_2) = \sum_{t \in [0, LCM]} eu(PB_1, t) \times eu(PB_2, t). \]

Now, the expected amount of parallelism between a processor $pr$ and all other processors in $PR$ is defined as the average expected amount of parallelism between the set of beads on $pr$ and the set of beads at each processor in $PR$. 

---

```
CalculateExpectedUtilization(in: PB: out: eu);
/* calculate the expected execution probability of each bead in PB */
for each b ∈ PB do
  for each t ∈ [0, LCM) do
    if b.eia ≤ t < b.eie
      then epet(b, t) := b.ddp × b.e/ (b.eie - b.eia)
      else epet(b, t) := 0;
    end for;
  end for;
/* calculate expected utilization based on the expected execution probabilities */
for each t ∈ [0, LCM) do eu(t) := ∑_{b∈PB} epet(b, t);
End.
```
**Definition 3.4 (Expected processor parallelism)** Let $B$ be the set of beads and $PR$ the set of processors. Let $pr \in PR$. Then, the expected amount of parallelism between processors is defined as

$$\text{eapp}(pr, PR) = \frac{1}{|PR| - 1} \times \sum_{pr' \in PR \setminus \{pr\}} \text{eap}(AB(pr'), AB(pr)).$$

The expected amount of parallelism as given by Definition 3.4 is calculated by the procedure in Figure 3.4. The procedure first calculates the expected

---

**Calculate_Expected_Parallelism**(inp, PR, B; out: eapp);
  eapp := 0;
  /* calculate the expected utilization for $AB(pr)$ */
  Calculate_Expected_Utilization($AB(pr)$, cu);
  for each $pr_1 \in PR \setminus \{pr\}$ do
    /* calculate the expected utilization for $AB(pr_1)$ */
    Calculate_Expected_Utilization($AB(pr_1)$, cu_1);
    /* calculate expected parallelism */
    eapp := 0;
    for each $i \in [0,|CM|)$ do
      eapp := eapp + cu_1(i) \times cu(i);
    end for;
    eapp := eapp/(|PR| - 1);
  end for;
End.

---

**Figure 3.4**: The procedure for calculating the expected amount of parallelism between a processor $pr$ and all other processors in $PR$.

---

utilization for the set of beads at processor $pr$ using the procedure in Figure 3.3. Then, for each other processor in $PR$ the expected utilization is calculated and the expected amount of parallelism with $pr$ is determined based on Definition 3.3. All these amounts are added up and divided by the amount of processors in $PR \setminus \{pr\}$ to obtain the average. The calculation of the expected amount of parallelism given by Definition 3.3 is done for each interval in which one of the utilizations is constant. Therefore, the entire procedure has a time-complexity $O(|PR| \times |B|)$.

### 3.2.4 Communication

The last system property discussed is the amount of communication of the set of beads $B$ on a communication medium $cm \in C'M$. The exact amount of communication on a medium is the amount of time the medium is accessed
by any of the beads in $B$. Therefore, the amount of communication is defined as follows:

**Definition 3.5 (Medium communication amount)** Let $B$ be the set of beads. Let $CM$ be the set of communication media and let $cm \in CM$. Then, the amount of communication over a medium is defined as

$$\text{acm}(B, cm) = \sum_{b \in B : b \cdot cm = cm} b \cdot e.$$ 

Now, the amount of time in $[0, \text{LCM})$ that is spend on communication is defined as the maximum amount of communication over all media as given by Definition 3.5.

**Definition 3.6 (Communication amount)** Let $B$ be the set of beads. Let $CM$ be the set of communication media. Then, the amount of communication is defined as

$$\text{act}(B, CM) = (\text{Max } cm \in CM : \text{acm}(B, cm)).$$

The amount of communication as given by Definition 3.6 is calculated by the procedure in Figure 3.5. This procedure calculates $\text{act}$ by calculating $\text{acm}$.

---

**Figure 3.5:** The procedure for calculating the amount of communication in a set of beads $B$ on a set of communication media $CM$.

for each medium in $CM$. The calculation of $\text{acm}$ is done by checking each bead in $B$. However, the procedure can be easily implemented such that it
checks each bead in $B$, while maintaining an $acm$ for each medium in $CM$. Then, the procedure has a time-complexity $O(|B|)$.

In the next section, a process assignment algorithm is presented which uses the communication and parallelism system properties defined in this section.

### 3.3 A Process Assignment Algorithm

In this section, a heuristic constructive algorithm is proposed that can solve the process assignment problem (see Figure 3.6). The algorithm tries to

```plaintext
Process Assignment Algorithm (in: $PS, B, RTC, CC, PR, CM, RO$; out: $FOUND, PA$):
/* Invariant: CorrectProcessAssignment($PA$) */
for each $ps \in PS$ do $PA(ps) := 0$;
/* Invariant holds \& $APS(PA) = 0$ */
while $¬(V ps \in PS : PA(ps) \in PR)$ do
  Select Process($PS, PA, ps$);
  /* $PA(ps) \notin PR$ */
  $FOUND := false$;
  $numberOfAttempts := 0$;
  /* $numberOfAttempts < MNA$ */
  while ($¬$ $FOUND \land numberOfAttempts < MNA$) do
    $numberOfAttempts := numberOfAttempts + 1$;
    /* try to find a processor spr for process $ps$ */
    Select Processor($ps, B, RTC, CC, PR, CM, RO, PA, FOUND, spr$);
    /* ($FOUND \land PA(ps) = spr$) \Rightarrow CorrectProcessor(spr, PA) */
    if $FOUND$ then
      /* assign spr to spr */
      $PA(spr) := spr$;
      /* CorrectProcessor(spr, PA) \land PA(spr) \in PR */
      /* Invariant holds */
      Insert Communication($ps, RO, B, RTC, CC$);
      /* $B$ is extended with communication beads and */
      /* RTC and CC are adjusted accordingly */
    else /* reassign a process */
      Reassign Process($PS, B, RTC, CC, PR, CM, RO, PA$);
      end if;
      /* Invariant holds */
    end while;
    /* $FOUND \Rightarrow PA(spr) \in PR$ */
    if $¬$ $FOUND$ then Exit;
  end while;
  /* CorrectProcessAssignment($PA$) \land $FOUND \Rightarrow APS(PA) \equiv PS$ */
End.
```

---

Figure 3.6: The process assignment algorithm.
find a process assignment $PA$ by assigning processes to processors one by one. The invariant of the algorithm is $CorrectProcessAssignment(PA)$ as defined by Property 2.2. The algorithm continues assigning processes until either a process can not be assigned or all processes are assigned. In the latter case, the process assignment $PA$ is such that $APS(PA) = PS$. Since $CorrectProcessAssignment(PA)$ is invariant, the process assignment problem is solved.

In order to assign a single process, the algorithm selects an unassigned process $sps \in PS$, i.e., for which $PA(sps) \notin PR$. Process selection is done by a procedure described in Section 3.3.2. Subsequently, the algorithm makes at most MNA attempts to find a processor to which process $sps$ can be assigned. If a processor is not found after MNA attempts, the algorithm terminates and the process assignment problem is not solved.

Per attempt, the algorithm tries to find a processor $spr$ that remains correct when process $sps$ is assigned to it. Thus, $CorrectProcessor(spr, PA)$, as defined by Property 2.1, remains satisfied when $PA(sps) = spr$. Processor selection is done by a procedure described in Section 3.3.3. This procedure assures that processor $spr$ is also "possibly feasible" when process $sps$ is assigned to it. The latter is determined by a feasibility condition and an objective function described in Section 3.3.3. If a processor is found, $sps$ is assigned to it and communication between process $sps$ and other already assigned processes is inserted. Communication insertion is done by a procedure described in Section 3.3.1. If a processor is not found, an already assigned process is reassigned. Process reassignment is done by a procedure described in Section 3.3.4. This procedure assures that the invariant of the algorithm remains satisfied.

### 3.3.1 Communication Insertion

When a process is assigned to a processor, it is known which method calls to other assigned processes require communication and which routes have to be used to pass parameter data for these calls. Communication between a process $sps$ and all other assigned processes is inserted using the procedure in Figure 3.7. This procedure checks each relative timing constraint. If a relative timing constraint is a precedence relation between a bead in process $sps$ and a bead in another assigned process and parameter data has to be passed, communication is inserted between the two beads. This is done as informally described at the end of Section 2.5.1, by generating a sequence of communication hops, where each hop consists of two beads as depicted in
**Chapter 3. Process Assignment**

**Insert Communication** (in: sps, RO; inout: B, RTC, CC);
for each \((b_1, a_0, b_0) \in RTC\) do
  if \((l_0 = \infty) \land (b_1.pr \neq b_0.pr) \land (sps \in \{b_1.ps, b_2.ps\}) \land (b_2.ds \neq 0)\) then
    /* communication has to be inserted between \(b_1\) and \(b_2\) */
    RTC := RTC \ \{\((b_1, a_0, b_0)\)\};
    ro := "route in RO from \(b_1.pr\) to \(b_2.pr\) with \(n\) hops";
    for each \((sp, cm, rp, ps) \in ro\) do
      /* make beads for communication hop */
      \(b^1 := \text{"sending bead of communication hop \(i\)"} ;
      b^2 := \text{"receiving bead of communication hop \(i\)"};
      b^1.ps := 0; b^2.ps := 0;
      b^1.ac := b_1.ac; b^2.ac := b_1.ac;
      b^1.cm := cm; b^2.cm := cm;
      b^1.c := ((\(b_1.ds/cm_i, dm\)) + 1) \times cm_i, bd;
      b^2.c := ((\(b_2.ds/cm_i, dm\)) + 1) \times cm_i, bd;
    /* insert relative timing constraint between the two beads */
    if \((cm_i, mod = "asynchronous")\) then
      RTC := RTC \ \{(b^1, b^2, e, 0, b_1), (b^2, b^1, e, 0, b_2)\}
    else
      RTC := RTC \ \{(b^1, 0, 0, b_1), (b^2, 0, 0, b_2)\};
    /* add consistency constraints with other beads */
    for each \(b \in B\) do
      if \((b.cm = cm_i \land b.ac \neq b^1.ac)\) then
        CC := CC \ \{(b, b^1, e, b^2)\};
      CC := CC \ \{(b, b^2, e, b^1)\};
    end if;
  end if;
end for;
/* add beads to \(B\) */
B := B \ \{b^1, b^2\};
end for;
/* insert precedence relations between separate hops and */
/* insert complete sequence between \(b_1\) and \(b_2\) */
RTC := RTC \ \{(b_1, b_2, e, \infty, b_1)\};
for each \(i : 1 \leq i < n\) do
  RTC := RTC \ \{(b^i, b^{i+1}, e, \infty, b^i)\};
end for;
RTC := RTC \ \{(b_n, b_1, e, \infty, b_2)\};
end if;
end for;
/* communication is inserted for process sps */
End.

**Figure 3.7:** The procedure that inserts communication between process sps and all other assigned processes using the set of routes RO. Thereby, the set of beads B is extended with communication beads, and the set of relative timing constraints RTC and the set of consistency constraints are extended accordingly.
3.3. A Process Assignment Algorithm

Figure 2.5. The two beads of a hop are either related by a precedence relation when the medium used in the hop is asynchronous or a relative timing constraint \((0,0)\) when this medium is synchronous. For each communication bead, a consistency constraint with each communication bead of another activity that accesses the same medium is added to \(CC\). Finally, all inserted communication beads are added to the set of beads \(B\).

The communication insertion procedure checks per element of \(RTC\) a route of which the number of hops is at most equal to the number of processors in \(PR\). Per hop of a route, each bead in \(B\) is checked for a consistency constraint. Therefore, the entire procedure has a time-complexity \(O(|RTC| \times |PR| \times |B|)\).

3.3.2 Process Selection

Process selection is done by the procedure in Figure 3.8. This procedure

```plaintext
Select_Process(in: PS, PA; out: sps);
    bestvalue := -\infty;
    for each \(ps \in PS\) do
        if \((PA(ps) \notin PR \land ps.e > bestvalue)\) then
            sps := ps;
            bestvalue := ps.e;
        end if;
    end for;
/* \(PA(sps) \notin PR \land (\forall ps \in PS : PA(ps) \notin PR : ps.e \geq ps.e)\) */
End.
```

Figure 3.8: The procedure that selects an unassigned process \(sps\) from \(PS\) based on the current process assignment \(PA\).

selects an unassigned process based on the heuristic that it is recommendable to first select a process that requires a large amount of execution time. The execution time of a process is defined as follows.

**Definition 3.7 (Process execution time)** Let \(PS\) be the set of processes and \(B\) the set of beads. Let \(ps \in PS\). Then, the execution time of a process is defined as

\[
ps.e = \sum_{b \in B : b.ps = ps} b.e.
\]
The set of beads that execute in a particular process does not change during process assignment. Thus, the execution time of a process is assumed to be given by the designer.

If a process with a large execution time is assigned last, there might not be a processor that can serve the process. Therefore, based on the current process assignment $PA$, the process selection procedure selects a process $sps \in PS$ for which the following holds:

$$PA(sps) \notin PR \land (\forall ps \in PS : PA(ps) \notin PR : \text{spse} \geq \text{ps.e}). \quad (3.3)$$

The process selection procedure has a time-complexity $O(|PS|)$.

### 3.3.3 Processor Selection

The processor selection procedure tries to select a processor $spr$ to which the selected process $sps$ can be assigned (see Figure 3.9). The first goal of the procedure is to find a processor $spr$ that remains correct when process $sps$ is assigned to it. However, recall that it is best to make a process assignment for which there is a feasible bead schedule. Unfortunately, deciding whether this holds is an NP-complete problem. Therefore, a feasibility condition for a processor is identified. In addition, an objective function is used to estimate the probability that a feasible bead schedule exists for a process assignment. The objective function is based on the system properties discussed in Section 3.2.

Usually, there are certain processors for which assignment obviously doesn't lead to a process assignment for which there is a feasible bead schedule and assignment of $sps$ to such a processor has to be avoided. Therefore, the processor selection procedure first constructs a set $CPR$ of candidate processors using a procedure described in Section 3.3.3.2. This is done such that the number of candidate processors is at most $NCPR$ and each candidate processor $pr$ remains correct when process $sps$ is assigned to it. Thus, $(PA(sps) = pr) \Rightarrow CorrectProcessor(pr, PA)$. When the set of candidate processors is constructed, the processor selection procedure checks each candidate using a feasibility condition and an objective function described below. Finally, the procedure selects the candidate processor $spr$ that satisfies the feasibility condition and has the best objective function value. Since $spr$ is a candidate, $(PA(sps) = spr) \Rightarrow CorrectProcessor(spr, PA)$. Thus, as required, a processor is selected that remains correct when $sps$ is assigned to it.
3.3. A Process Assignment Algorithm

Select Processor (in: sps, B, RTC, CC, PR, CM, RO, PA; out: FOUND, spr);
/* initialize FOUND and best objective function value */
spr := 0; bestvalue := oc;
/* construct a set CPR of at most NCPR candidate processors */
ConstructCandidateProcessorSet(sps, B, RTC, CC, PR, PA, CPR);
/* (forall pr ∈ CPR : PA(sps) = pr ⇒ CorrectProcessor(pr, PA)) */
/* check assignment of sps to each candidate processor */
forall pr ∈ CPR do
  /* create situation in which sps is assigned to pr */
  PA(sps) := pr;
  InsertCommunication(sps, RO, B, RTC, CC);
  /* calculate execution intervals based on new assignment */
  CalculateExecutionIntervals(B, RTC, CC, FOUND);
  /* only continue if execution intervals are calculated */
  if FOUND then
    /* check feasibility condition for processor pr */
    CheckFeasibilityCondition(pr, B, FOUND);
    /* check new assignment only if condition holds */
    if FOUND then
      /* calculate objective function for processor pr */
      CalculateExpectedParallelism(pr, CPR, B, capp);
      CalculateCommunication(B, CM, act);
      prof := ω_e × act − ω_p × capp;
      /* check if pr has minimum objective function value */
      if (prof < bestvalue) then spr := pr; bestvalue := prof;
    end if;
  end if;
end for;
/* undo situation in which sps is assigned to pr */
DeleteCommunication(sps, B, RTC, CC);
PA(sps) := 0;
FOUND := (spr ∈ CPR);
/* FOUND ⇒ (PA(sps) = spr ⇒ CorrectProcessor(spr, PA)) */
End.

Figure 3.9: Procedure for selecting a processor spr for a process sps.

A candidate processor pr ∈ CPR is checked by creating a situation in which sps is temporarily assigned to pr. Thereby, communication from sps to other assigned processes is inserted using the procedure of Figure 3.7. Then, execution intervals are calculated based on this new set of beads and constraints using the procedure of Figure 3.2. Since communication beads are inserted, the amount of execution time may become too large to satisfy the absolute timing constraints. In this case, execution intervals cannot be calculated. If execution intervals can be calculated, it is checked whether the beads at pr can still be scheduled by evaluating a feasibility condition discussed in Section 3.3.3.1. Only if this feasibility condition is satisfied, a
processor is found. Then, the procedure continues with calculating the processor selection objective function \( psorf \) for processor \( pr \) as given below by Definition 3.8. Finally, process \( sps \) is removed from processor \( pr \) and communication from \( sps \) to other assigned processes is deleted using a procedure described in Section 3.3.3.3.

The objective function reflects that a feasible bead schedule probably exists for process assignment \( PA \) when (1) the amount of communication is minimal and (2) the amount of parallelism is maximal. The objective function uses Definition 3.6 for the amount of communication and Definition 3.4 for the expected amount of parallelism for a processor \( pr \in CPR \).

**Definition 3.8 (Processor selection objective function)** Let \( B \) be the set of beads. Let \( CM \) be the set of communication media and \( PR \) the set of processors. Let \( CPR \subseteq PR \) be a set of candidate processors. Let \( pr \in CPR \). Then, the processor selection objective function is defined as

\[
psorf(pr, CPR, B, CM) = \omega_c \times act(B, CM) - \omega_p \times eapp(pr, CPR).
\]

\[ \square \]

Minimizing communication and maximizing parallelism are two opposite objectives. The first objective tends to cluster all processes onto a single processor while the second tends to evenly spread the processes. Therefore, in \( psorf \) the expected amount of parallelism is subtracted from the amount of communication. In addition, the communication and parallelism objectives are weighted with weights \( \omega_c \) respectively \( \omega_p \). Thus, the processor selection procedure can be guided by tuning the weights of the objective function. A processor is better when the parallelism is larger and the communication is smaller, i.e., the objective function value is smaller.

**3.3.3.1 Feasibility Condition**

The feasibility condition used in the processor selection procedure is equivalent to the feasibility condition defined in Property 2.15 for a single processor. Here, the feasibility condition is expressed in terms of execution intervals instead of windows. Thus, for a processor \( pr \), the following must hold:

\[
(\forall b_1, b_2 \in AB(pr) : b_1, eis < b_2, eis : \\
(\sum b_3 \in AB(pr) : b_1, eis \leq b_3, eis \wedge b_3, eic \leq b_2, eic : b_3, e) \\
\leq b_2, eic - b_1, eis
).
\]

(3.4)
3.3. A Process Assignment Algorithm

This feasibility condition is evaluated by the procedure in Figure 3.10. The

\[
\text{Check\_Feasible\_Condition}(\text{inpr}, B; \text{out}_F\text{FOUND});
\]

\[
\text{FOUND} := \text{true};
\]

\[
\text{SL}_B := \text{"list of beads } b \in AB(\text{pr}) \text{ sorted by non-decreasing } b.eis";
\]

for each \( b_1 \in AB(\text{pr}) \) do

\[
\text{b}_2 := \text{"first bead in } \text{SL}_B \text{ with } b_1.eis < b_2.eis";
\]

\[
\text{sum} := 0;
\]

while "there is a \( b_2 \)" do

\[
\text{if } (b_1.eis \leq b_2.eis) \text{ then }
\]

\[
\text{sum} := \text{sum} + b_2.e;
\]

\[
/* \text{sum} := (\sum_{b_3 \in AB(\text{pr}) : b_3.eis \leq b_2.eis} b_3.e) */
\]

\[
\text{FOUND} := \text{FOUND} \land (\text{sum} \leq b_2.eis - b_1.eis);
\]

end if;

\[
b_2 := \text{"next bead after } b_2 \text{ in } \text{SL}_B";
\]

end while;

end for;

\[
/* \text{FOUND } \Leftrightarrow (3.4) \text{ is satisfied } */
\]

End.

Figure 3.10: The procedure that checks feasibility condition (3.4) for a processor pr and a set of beads B.

The procedure first sorts all beads on pr by non-decreasing execution interval end time. This is done by a standard sorting procedure that has a time-complexity less than \( O(|B|^2) \). Then, for each execution interval start time, the procedure passes forwards through the sorted list of beads only once, while maintaining and checking the execution sum. Therefore, the entire procedure has a time-complexity \( O(|B|^2) \).

3.3.3.2 Candidate Processor Set Construction

The candidate processor set for a process is constructed by the procedure in Figure 3.11. This procedure first calculates execution intervals for the current set of beads using the procedure of Figure 3.2. Then, per processor in \( PR \), the procedure checks if the processor remains correct and if a constraint on the relative timing constraints is satisfied when process \( sps \) is assigned to it. Finally, the set \( CPR \) is composed of at most \( NCPR \) processors that remain correct and satisfy the constraint.

A processor \( pr \in PR \) remains correct when process \( sps \) is assigned to it.
Construct_Candidate_Processor_Set (in: sps, B, RTC, CC, PR, PA; out: CPR);
   CPR := ∅;
   /* calculate execution intervals based on current assignment */
   Calculate_Execution_Intervals(B, RTC, CC, FOUND);
   /* exit when execution intervals could not be found */
   if ¬(FOUND) then Exit;
   for each pr ∈ PR do
      nd := pr.md;
      for each ps ∈ APS(pr, PA) do
         nd := nd + ps.nd;
         correct := nd ≤ pr.md;
         /* correct ⇔ (3.5) is satisfied */
         for each (b₁, s₀, lo, b₂) ∈ RTC do
            if (sps ∈ {b₁, p₁, b₂, ps}) ∧ (pr ∈ {b₁, pr, b₂, pr})
               then correct := correct ∧ lo ≥ b₁;
         end for;
         /* correct ⇔ (3.5) and (3.6) is satisfied */
         if (correct) then CPR := CPR ∪ {pr};
      end for;
      /* CPR is the set of processors that satisfy (3.5) and (3.6) */
      while (|CPR| > N CPR) do
         pr := "a random element from CPR";
         CPR := CPR \ {pr};
      end while;
      /* CPR contains at most N CPR elements */
      /* (∀ pr ∈ CPR : PA(pr) = pr ⇒ Correct_Processor(pr, PA)) */
   End.

Figure 3.11: The procedure for constructing a set of candidate processors CPR for a process sps.

if the following holds based on the current process assignment PA:

\[
\text{sps entrances} + \sum_{ps\in APS(pr, PA)} ps.\text{nd} \leq pr.\text{md} \quad (3.5)
\]

This condition expresses that the number of devices managed by sps plus the number of devices already assigned to pr does not exceed the maximum number of devices that can be connected to pr. Thus, \((PA(ps) = pr) \Rightarrow Correct\Processor\,(pr, PA)\). This condition is checked by adding the number of devices of all the processes assigned to pr. Therefore, this part of the procedure has a time-complexity \(O(|PS|)\).

The constraint used for candidate set construction expresses that the relative timing constraints between beads of process sps and beads on processor pr may not be too tight:

\[
(∀ (b₁, s₀, lo, b₂) \in RTC : \quad \text{constraint holds})
\]
3.3. A Process Assignment Algorithm

\[ sps \in \{b_1.ps, b_2.ps\} \land pr \in \{b_1.pr, b_2.pr\} : lo \geq b_1.e \]

If this constraint is not satisfied for a relative timing constraint between \(b_1\) and \(b_2\), beads \(b_1\) and \(b_2\) cannot be scheduled on the same processor without overlap in time. This is evaluated by checking each relative timing constraint. Therefore, this part of the procedure has a time-complexity \(O(|RTC|)\).

Summarizing, the part of the procedure that checks (3.5) and (3.6) for each processor has a time-complexity \(O(|PR| \times (|PS| + |RTC|))\). The entire candidate set construction procedure has a time-complexity \(O(|B| + |RTC|^2 \times |CC| + |PR| \times (|PS| + |RTC|))\).

### 3.3.3.3 Communication Deletion

When a process is removed from a processor, the communication between the process and all other assigned processes has to be deleted. This is done by the procedure in Figure 3.12. This procedure passes through the execution

```
Delete_Communication(inp:ps, inout:B, RTC, CC);
  for each \(b_1 \rightarrow b_n\) do
    if \((ps \in \{b_1.ps, b_n.ps\}) \land (\forall i : 2 \leq i \leq n-1 : b_i.cm \neq 0)\) then
      /* communication has to be deleted between \(b_1\) and \(b_n\) */
      RTC := RTC \ {(b_1, b_i.e, oo, b_n)};
      for each \(i : 2 \leq i \leq n-1\) do
        /* delete precedence relation with successor */
        RTC := RTC \ {(b_i, b_(i+1).e, oo, b_n)}
      /* delete consistency constraints with other beads */
      for each \(\alpha \in CC\) do
        if \((b_i \in \alpha)\) then CC := CC \ \{\alpha\};
      end for;
      /* delete bead from \(B\) */
      B := B \ \{b_i\};
    end for;
    /* insert precedence relation between \(b_1\) and \(b_n\) */
    RTC := RTC \ {(b_1, b_1.e, oo, b_n)};
  end if;
end for;
/* communication is deleted for process \(sp\) */
End.
```

**Figure 3.12:** The procedure that deletes communication between process \(sp\) and all other assigned processes.

DAG of beads in a forward direction. Thereby, the procedure repeatedly
detects a path $b_1 \rightarrow \ldots \rightarrow b_n$ in which either $b_1$ or $b_n$ is a bead from process $sps$ and all beads $b_2$ to $b_{n-1}$ are communication beads. Then, all consistency constraints and relative timing constraints in which a communication bead of the path is involved are removed from sets $CC$ and $RTC$ respectively. In addition, all communication beads on the path are deleted from $B$ and a precedence relation from $b_1$ to $b_n$ is added to $RTC$.

The communication deletion procedure detects all communication paths by passing once through the execution DAG. Per detected path, the procedure checks all consistency constraints in $CC$. Therefore, the entire communication deletion procedure has a time-complexity $O(|RTC| \times |CC|)$.

Now, the time-complexity of the entire processor selection procedure can be determined. Note that the for-loop of the processor selection procedure, as depicted in Figure 3.9, is executed for a constant number of processors. Thus, the time-complexity of the entire procedure can be determined by adding the time-complexities of all the separate procedures. It can be verified that the processor selection procedure has a time-complexity $O(|RTC|^2 \times |CC| + |PR| \times (|PS| + |RTC| \times |B|) + |B|^2)$.

### 3.3.4 Process Reassignment

When a processor is not found by the processor selection procedure, the process assignment algorithm reassigns a process. Process reassignment is done by the procedure in Figure 3.13. If reassignment is necessary, the execution time at each processor is so high that no new process can be assigned. Therefore, the current process assignment is slightly changed by reassigning a single process. Thereby, the load on the processor to which the process is currently assigned is reduced. Of course, the processor to which the process is reassigned may not become overloaded. A processor is defined not to be overloaded if the sum of the execution times of all the beads assigned to the processor is at most the length LCM of the entire execution interval. This is expressed by the following property.

**Property 3.1** Let $PB$ be a set of beads assigned to the same processor. Then, the processor not overloaded property for set $PB$ is defined as

$$
\text{ProcessorNotOverloaded}(PB) \iff \sum_{b \in PB} b.r \leq \text{LCM}.
$$
Reassign_Process((in:PS,B,RTC,CC,PR,CM,RO; inout:PA);
/* select a least full processor cpr */
bestvalue := oo;
for each pr ∈ PR do
  if (APS(pr, PA) ≠ ∅) then
    sum := 0;
    for each b ∈ AB(pr) do sum := sum + b.e;
    if (sum < bestvalue) then
      cpr := pr; bestvalue := sum;
    end if;
  end if;
end for;
/* cpr is a least full processor */
bestvalue := oo;
for each ps ∈ APS(cpr, PA) do
  if (ps.e < bestvalue) then
    rps := ps; bestvalue := ps.e;
  end if;
end for;
/* rpa is smallest process on cpr */
Delete_Communication(rps, B, RTC, CC);
P.A(rps) := 0;
/* rpa is removed from cpr and communication involved is deleted */
bestvalue := oo;
for each pr ∈ PR \ {cpr} do
  nd := rps.nd;
  for each ps ∈ APS(pr, PA) do nd := nd + ps.nd;
  correct := nd ≤ pr.md;
  /* correct ⇔ (3.5) is satisfied */
  for each (b1, a1, lo, b2) ∈ RTC do
    if (rps ∈ {b1, ps, b2, ps}) ∧ pr ∈ {b1, pr, b2, pr})
      then correct := correct ∧ lo ≥ b1.e;
  end for;
  /* correct ⇔ (3.5) and (3.6) is satisfied */
  sum := 0;
  for each b ∈ AB(pr) do sum := sum + b.e;
  correct := correct ∧ rps.e + sum ≤ LCM;
  if (correct ∧ sum < bestvalue) then
    rpr := pr; bestvalue := sum;
  end if;
end for;
/* P.A(rps) = rpr ⇒ CorrectProcessor(rpr, PA) */
P.A(rps) := rpr;
Insert_Communication(rps, AC, B, RTC, CC);
/* rpa is assigned to rpr and communication is inserted accordingly */
/* CorrectProcessAssignment(PA) */
End.

Figure 3.13: The procedure that reassigns a single process.
The reassignment heuristic tries to reduce the load on the least full processor. Therefore, the reassignment procedure selects a least full processor \( pr \) among all processors to which at least one process is assigned according to

\[
(\forall pr \in PR : \sum_{b \in AB(pr)} b.e \geq \sum_{b \in AB(cpr)} b.e).
\]

This can be implemented by a loop over all beads, while the total execution time per processor is maintained. Then, the processor with the minimum sum is selected. Thus, this part of the procedure has a time-complexity \( O(|B| + |PR|) \).

Then, the load on the least full processor is reduced by the smallest amount possible. Therefore, the procedure selects a process \( rps \) with the smallest total execution time among all processes assigned to \( pr \)

\[
(\forall ps \in PS : PA(ps) = pr : ps.e \geq rps.e).
\]

Since the execution time of a process is given by the designer, this part of the procedure has a time-complexity \( O(|PS|) \). Process \( rps \) is removed from processor \( pr \) and all communication between \( rps \) and other assigned processes is deleted using the procedure of Figure 3.12.

Now, process \( rps \) must be reassigned to a processor that (1) remains correct as described by (3.5), (2) satisfies constraint (3.6) and (3) does not become overloaded as given by Property 3.1. For convenience, the set \( PP(rps) \) of possible processors to which \( rps \) can be reassigned is defined as

\[
PP(rps) = \{ pr \in PR \setminus \{ pr \} : \sum_{ps \in ASS(ps, PA)} ps.nd \leq pr.md \} \land
(\forall (b_1, s_0, lo, b_2) \in RTC : rps \in \{ b_1, ps, b_2, ps \} \land pr \in \{ b_1, pr, b_2, pr \} : lo \geq b_1.e \land Processor:\text{NotOverloaded}(AB(pr) \cup \{ b \mid b.ps = rps \}) \}
\].
Then, the least full processor \( rpr \) is selected from \( PP(rps) \)

\[
(\forall pr \in PP(rps) : \sum_{b \in \mathcal{A}B(pr)} b.e \geq \sum_{b \in \mathcal{A}B(rpr)} b.e).
\]

This can be implemented in a loop over all processors, while per processor (3.5) and (3.6) is checked and the execution sum is calculated in a loop over all beads in \( B \). Therefore, this part of the procedure has a time-complexity \( \mathcal{O}(|PR| \times (|PS| + |RTC| + |B|)) \).

Finally, process \( rps \) is assigned to processor \( rpr \) and communication between \( rps \) and other assigned processes is inserted using the procedure of Figure 3.7. Note that, since \( rpr \) is a processor from \( PP(rps) \), \( rpr \) remains correct when process \( rps \) is assigned to it, i.e., \( (PA(rps) = rpr) \Rightarrow CorrectProcessor(rpr, PA) \). Consequently, the process assignment also remains correct.

Now, the time-complexity of the entire procedure can be determined by adding the time-complexities of all the separate parts. It can be verified that the procedure for process reassignment has a time-complexity \( \mathcal{O}(|RTC| \times (|PR| \times |B| + |CC|) + |PR| \times |PS|) \).

Now, the time-complexity of the entire process assignment algorithm can be determined. The inner while-loop of the algorithm, as depicted in Figure 3.6, is executed a constant number of times. The outer while-loop of the algorithm is executed at most for each process in \( PS \). The time-complexity of one iteration of the outer while-loop can be determined by adding the time-complexities all separate procedures. It can be verified that the time-complexity of the processor selection procedure dominates the time-complexity of all other procedures. Thus, the time-complexity of the entire process assignment algorithm equals the time-complexity of the processor selection procedure times the number of processes. Therefore, the process assignment algorithm has a time-complexity \( \mathcal{O}(|PS| \times (|RTC|^2 \times |CC| + |PR| \times (|PS| + |RTC| \times |B|) + |B|^2)) \). Results of performance measurements of the algorithm are presented and discussed in Chapter 6.

### 3.4 Conclusion

In this chapter, a heuristic constructive algorithm is presented that can solve the process assignment problem of first step of the approach. The algorithm assigns processes one by one to processors in order of decreasing execution...
time. In order to obtain a possibly feasible process assignment, the algorithm uses a feasibility condition and an objective function. The objective function is based on several system properties that are defined in terms of probabilistic formulas. The time-complexity of the algorithm is acceptable for pre-runtime scheduling. Results of performance measurements of the algorithm can be found in Chapter 6.
Chapter 4

Window Assignment

In this chapter, a window assignment algorithm is presented that can solve the window assignment problem of the second step of the approach. The window assignment algorithm first transforms consistency constraints into relative timing constraints with heuristics that attempt to minimize the subsequent adjustment to absolute timing constraints. During consistency constraint transformation, the algorithm checks if the feasibility conditions can still be satisfied. Then, a window is assigned to each bead by a heuristic that attempts to spread the execution time evenly over the most critical path. This path is defined using a metric for the contention of beads at the same processor. After assignment of a window to a bead, the windows of other beads are adjusted or shifted such that the feasibility conditions can still be satisfied. Window shifting in this way turns out to be a useful technique to reduce the amount of backtracking during window assignment. This is supported by performance measurements of the window assignment algorithm presented in Chapter 6. The results also show that there is no backtracking at all during the subsequent local scheduling step.

4.1 The Window Assignment Problem

The window assignment problem concerns a set of beads $B$ already assigned to a set of processors $PR$. The execution of the beads is restricted by (1) absolute timing constraints, (2) a set of relative timing constraints $RTC$ and
(3) a set of consistency constraints $CC$. A window has to be assigned to each bead in $B$, such that all constraints are satisfied when each bead is scheduled in its window. As defined in Section 2.6.2, the window assignment problem is to find a presumably feasible window assignment $WA$ in which a window $[WA(b).ws, WA(b).we]$ is assigned to each bead $b$ in $B$. Thus, $PresumablyFeasibleWindowAssignment(WA, B, PR, RTC, CC)$ must hold. As discussed in Section 2.6.2, this problem is probably NP-complete.

In this chapter, a constructive heuristic algorithm is presented that can find a presumably feasible window assignment. For this purpose, the problem is first analyzed and a polynomial time algorithm is presented that always finds a correct window assignment for a set of beads without consistency constraints. Then, two extension are made to this algorithm to incorporate consistency constraints and feasibility conditions in order to solve the entire window assignment problem.

### 4.2 Problem Analysis

A correct window assignment for a set of beads with absolute timing constraints and a set of relative timing constraints $RTC$ but without consistency constraints is a window assignment $WA$ for which $CorrectWindowAssignment(WA, B, RTC, \emptyset)$ holds. The problem of finding such a window assignment is not NP-complete, because it can be solved in polynomial time by the algorithm in Figure 4.1. This algorithm has a time-complexity that is polynomial in the number of beads and the number of relative timing constraints. The algorithm repeatedly selects a bead and assigns a window to it. Thereby, the algorithm maintains a set of beads $WB$ to which windows are already assigned. The invariant of the algorithm is $CorrectWindowAssignment(WA, WB, RTC, \emptyset)$. Initially $WB = \emptyset$ and thus the invariant is satisfied. When a window is assigned to each bead, i.e., $WB = B$, $CorrectWindowAssignment(WA, B, RTC, \emptyset)$ holds and the problem is solved.

The main concepts of the algorithm are described first, while the detailed structure of the algorithm is explained later. As discussed above, the window assignment $WA$ that has to be found must satisfy $CorrectWindowAssignment(WA, B, RTC, \emptyset)$, i.e., Properties 2.9 to 2.11 must be satisfied by the window assignment. A closer examination of these properties reveals that the window start time of a bead must be in a certain interval. This interval is based on the absolute timing constraints of the bead, but also on the
4.2. Problem Analysis

```c
Assign_Correct_Windows(in B, RTC; out:FOUND,WA):
  /* WB is the set of beads to which windows are already assigned */
  /* Invariant: CorrectWindowAssignment(WA, WB, RTC, #) */
  WB := #;
  /* Invariant holds */
  /* Initialize each correct interval as large as possible (see Definition 4.1) */
  for each b ∈ B do
    b.scwa := b.est; b.lcws := b.d - b.e;
    b.scwe := b.est + b.e; b.lcwe := b.d;
  end for;
  /* All correctness constraints hold except Property 4.3 */
  Adjust_Correct_Interval_To_Correctness_Constraints(B, RTC, FOUND);
  /* FOUND = all correctness constraints hold */
  while FOUND ∧ (WB ≠ B) do
    /* Loop invariant: all correctness constraints hold */
    wb := Random_Element(B \ WB);
    WA(wb).ws := Random_Value([wb.scwa, wb.lcwa]);
    WA(wb).we := Random_Value([WA(wb).ws + wb.e, WA(wb).ws + wb.lwe]);
    /* See for WA Definition 2.15 and for lwu Definition 4.2 */
    WB := WB ∪ {wb};
    wb.scwa := WA(wb).ws; wb.lcws := WA(wb).ws;
    wb.scwe := WA(wb).we; wb.lcwe := WA(wb).we;
    /* All correctness constraints hold except Property 4.3 */
    Adjust_Correct_Interval_To_Correctness_Constraints(B, RTC, FOUND);
    /* FOUND = true and thus invariant holds */
  end while;
  /* FOUND = CorrectWindowAssignment(WA, B, RTC, #) */
End.
```

Figure 4.1: An algorithm that finds a correct window assignment WA for a set of beads B with a set of relative timing constraints RTC if one exists.

window end times of the bead itself and of predecessor and successor beads. Similarly, the window end time of a bead must also be in a certain interval. This interval is based on the absolute timing constraints of the bead and on the window start times of the bead itself and of predecessor and successor beads. Therefore, the algorithm maintains for each bead a so-called correct window start time interval and a correct window end time interval. Then, assignment of a window to a bead is done by selecting a window start time and a window end time from the correct intervals of the bead. Subsequently, the correct intervals of other beads are adjusted to incorporate the effects of the assignment. In order to explain this in detail, the correct intervals are formally defined as follows.

**Definition 4.1 (Correct intervals)** Let B be the set of beads. Then, for each bead \( b ∈ B \) a correct window start time interval \([b.scwa, b.lcws]\)
and a correct window end time interval \([b.scwe, b.lcwe]\) is defined. Thus, \(b.scws \in \mathbb{N}\) is the smallest correct window start time and \(b lcws \in \mathbb{N}\) is the largest correct window start time. Similarly, \(b.scwe \in \mathbb{N}\) is the smallest correct window end time and \(b lcwe \in \mathbb{N}\) is the largest correct window start time. A correct interval must be non-empty

\[
(\forall b \in B : b.scws \leq b lcws \land b.scwe \leq b lcwe).
\]

Note that the window start time of a bead \(b\) is selected from its correct window start time interval, \(b.scws \leq b.ws \leq b lcws\), and the window end time of the bead is selected from its correct window end time interval, \(b.scwe \leq b.we \leq b lcwe\).

Now, Properties 2.9 to 2.11 that must be satisfied by the window assignment are translated into properties that must be satisfied by the correct intervals. First, Property 2.9 expresses that the window end time of a bead must be at least the window start time of the bead plus the execution time of the bead. Thus, the smallest correct window end time of a bead must also be at least the smallest correct window start time of the bead plus the execution time of the bead. In addition, a similar constraint must be satisfied by the largest correct values. This is expressed by the following property.

**Property 4.1** Let \(B\) be the set of beads. Then, the minimum correct interval distance property for the beads in \(B\) is defined as

\[
\text{MinimumCorrectIntervalDistance}(B) \quad \iff \quad (\forall b \in B : b.scws + b.e \leq b.scwe \land b lcws + b.e \leq b lcwe).
\]

Second, Property 2.10 implies that the correct intervals of a bead must be between the earliest start time and deadline of the bead.

**Property 4.2** Let \(B\) be the set of beads. Then, the absolute timed correct intervals property for the beads in \(B\) is defined as

\[
\text{AbsoluteTimedCorrectIntervals}(B) \quad \iff \quad (\forall b \in B : b.est \leq b.scws \land b lcwe \leq b.d).
\]
4.2. Problem Analysis

Note that this property in conjunction with Property 4.1 implies that the correct intervals of each bead are between the earliest start time and deadline of the bead.

Third, Property 2.11 expresses that for each relative timing constraint from a bead $b_1$ to a bead $b_2$, there must be a minimum distance between the window end time of $b_1$ and the window start time of $b_2$ and a maximum distance between the window start time of $b_1$ and the window end time of $b_2$. Obviously, this must also hold for the smallest values of the correct intervals of the beads and for the largest values. This is expressed by the following property.

**Property 4.3** Let $B$ be the set of beads and $RTC$ the set of relative timing constraints on the beads in $B$. Then, the relative timed correct intervals property for $B$ and $RTC$ is defined as

\[
\text{RelativeTimedCorrectIntervals}(B, RTC) \Rightarrow
\]

\[
(\forall b_1, b_2 \in B, so, lo \in \mathbb{N} : (b_1, so, lo, b_2) \in RTC : \]

\[
b_1.scwe - b_1.e + so \leq b_2.scws \land
\]

\[
b_2.scwe - b_2.e \leq b_1.scws + lo \land
\]

\[
b_1.lcwe - b_1.e + so \leq b_2.lcws \land
\]

\[
b_2.lcwe - b_2.e \leq b_1.lcws + lo
\].

\]

The correct intervals are initialized as large as possible according to Properties 4.2 and 4.1. Assignment of a window to a bead $b$ is done by selecting a window start time $WA(b).ws$ from the correct window start time interval $[b.scws, b.lcws]$ and a window end time $WA(b).we$ from the correct window end time interval $[b.scwe, b.lcwe]$. This must be done such that the width of the window is at least the execution time of the bead. Otherwise, Property 2.9 can not be satisfied. In addition, however, a largest window width $b.lw$ can be defined for each bead $b \in B$, based on the absolute and relative timing constraints.

**Definition 4.2 (Largest window width)** Let $B$ be the set of beads and $RTC$ the set of relative timing constraints on the beads in $B$. Then, the
Adjust_Correct_Interval_To_Correctness_Constraints (in: B, RTC; out: FOUND);
  FOUND := true; changed := true;
  while changed do
    changed := false;
    /* adjust correct intervals in forward direction */
    R := ∅; /* R is the set of ready beads */
    C := {b ∈ B | ¬(∃ b', so, lo : (b', so, lo, b) ∈ RTC)};
    /* C is the set of candidate beads */
    while C ≠ ∅ do
      b1 := "a random element from C";
      C \ {b1}; R ∪ {b1};
      if b1.acue < b1.acue + b1.c then
        b1.acue := b1.acue + b1.c; changed := true;
      if (b1.acue > b1.iave) then FOUND := false; Exit;
      for each b2 ∈ B : (b2.so, lo, b2) ∈ RTC do
        if b2.acue < b2.acue - b2.c + so
          then b2.acue := b2.acue - b2.c + so; changed := true;
        if b2.iave > b2.iave + b2.c + lo
          then b2.iave := b2.iave + b2.c + lo; changed := true;
          if (b2.acue > b2.iave) ∨ (b2.iave > b2.iave) then FOUND := false; Exit;
      if (∀ b' : (b', so', lo', b') ∈ RTC : b' ∈ R ∪ C) then C ∪ {b2};
    end while;
    /* adjust correct intervals in backward direction */
    R := ∅; C := {b ∈ B | ¬(∃ b', so, lo : (b', so, lo, b) ∈ RTC)};
    while C ≠ ∅ do
      b2 := "a random element from C";
      C \ {b2}; R ∪ {b2};
      if b2.iave > b2.iave - b2.c then
        b2.iave := b2.iave - b2.c; changed := true;
      if (b2.acue > b2.iave) then FOUND := false; Exit;
      for each b1 ∈ B : (b1.so, lo, b1) ∈ RTC do
        if b1.acue < b1.acue - b1.c - lo
          then b1.acue := b1.acue - b1.c - lo; changed := true;
        if b1.iave > b1.iave + b1.c - so
          then b1.iave := b1.iave + b1.c - so; changed := true;
          if (b1.acue > b1.iave) ∨ (b1.iave > b1.iave) then FOUND := false; Exit;
      if (∀ b' : (b', so', lo', b') ∈ RTC : b' ∈ R ∪ C) then C ∪ {b1};
    end while;
  end while;

End.

Figure 4.2: The procedure that adjusts the correct intervals of a set of beads B to the correctness constraints given a set of relative timing constraints RTC.
4.2. Problem Analysis

execution DAG followed by a backward pass is called a pass of the procedure. During a pass, Property 4.1 is checked for each bead and Property 4.3 is checked for each relative timing constraint. Therefore, a pass has a time-complexity $O(|\mathcal{B}| + |\mathcal{RTC}|)$. If correct intervals that satisfy all correctness constraints can be found, the number of relative timing constraints for which Property 4.3 is satisfied increases with at least one during each pass. Thus, the procedure can be terminated after a pass in which this number has not increased. Consequently, the number of passes is at most equal to $|\mathcal{RTC}|$ and the procedure has a time-complexity $O(|\mathcal{RTC}| \times (|\mathcal{B}| + |\mathcal{RTC}|))$.

Now, it can be proven that the procedure in Figure 4.2 can always adjust the correct intervals such that all correctness constraints are satisfied when it is called from within the while-loop of the algorithm of Figure 4.1. Note that window assignment and correct interval adjustment can be viewed as repeatedly reducing a correct interval of a bead to a subinterval and either (1) adjusting the other correct interval of the bead to satisfy Property 4.1 or (2) adjusting a correct interval of a bead involved in a relative timing constraint to satisfy Property 4.3.

In the following, Theorems 4.1 and 4.2 state that adjustment is always possible such that each correct interval remains non-empty provided that Property 4.1 and Property 4.3 are satisfied before reduction. The proofs of Theorems 4.1 and 4.2 do not depend on the exact value of the lower bound and upper bound of the subinterval to which a correct interval is reduced. This implies that any window can be assigned to a bead as long as the window start and end time are selected from the correct intervals and the width of the window is correct. In addition, windows can be assigned to beads in any order. Finally, based on Theorems 4.1 and 4.2 it can be concluded that correct interval adjustment in the while-loop of the algorithm of Figure 4.1 can always be performed, because all correctness constraints hold before the assignment of a window.

**Theorem 4.1** Let $\mathcal{B}$ be the set of beads and assume MinimumCorrect IntervalDistance($\mathcal{B}$) holds. Let $b \in \mathcal{B}$ and suppose one of the correct intervals of $b$ is reduced to a subinterval. Then, the other correct interval of $b$ can be adjusted to resatisfy MinimumCorrect IntervalDistance($\mathcal{B}$) such that the correct interval remains non-empty.

**Proof** In order to prove this theorem, two cases must be distinguished. In the first case, assume that the correct window start time interval of $b$ is reduced to $[b^w.sews, b^w.lews]$. Thus

$$b.sews \leq b^w.sews \leq b^w.lews \leq b.lews$$ (4.1)
holds. In order to resatisfy \( \text{MinimumCorrectIntervalDistance}(B) \), only \( b.\text{scwe} \) must be set to \( \text{Max}(b.\text{scwe}, b^n.\text{scws} + b.e) \). Then, the correct window end time interval of \( b \) remains non-empty when \( \text{Max}(b.\text{scwe}, b^n.\text{scws} + b.e) \leq b.\text{lcwe} \) holds. This can be proven as follows:

\[
\text{Max}(b.\text{scwe}, b^n.\text{scws} + b.e) \leq b.\text{lcwe} \\
\leftrightarrow \quad \{ \text{Calculus} \}
\]

\[
b.\text{scwe} \leq b.\text{lcwe} \land b^n.\text{scws} + b.e \leq b.\text{lcwe}
\leftrightarrow \quad \{ \text{Definition 4.1} \}
\]

\[
b^n.\text{scws} + b.e \leq b.\text{lcwe}
\leftrightarrow \quad \{ \text{Relation (4.1)} \}
\]

\[
b.\text{lcws} + b.e \leq b.\text{lcwe}
\leftrightarrow \quad \{ \text{Property 4.1} \}
\]

\[
\text{MinimumCorrectIntervalDistance}(B) \text{ before reduction} \leftrightarrow \quad \{ \text{Theorem assumption} \}
\]

\[\text{true.}\]

In the second case, assume that the correct window end time interval of \( b \) is reduced. Then, similar reasoning as in the first case can be used to show that the correct window start time of \( b \) remains non-empty when \( \text{MinimumCorrectIntervalDistance}(B) \) is resatisfied.

\[\square\]

**Theorem 4.2** Let \( B \) be the set of beads and \( RTC \) the set of relative timing constraints on the beads in \( B \). Assume \( \text{RelativeTimedCorrectIntervals}(B, RTC) \) holds. Let \( (b_1, so, lo, b_2) \in RTC \) and suppose a correct interval of \( b_1 \) or \( b_2 \) is reduced to a subinterval. Then, the other correct intervals of \( b_1 \) and \( b_2 \) can be adjusted to resatisfy \( \text{RelativeTimedCorrectIntervals}(B, RTC) \) such that these correct intervals remain non-empty.

**Proof** In order to prove this theorem, four cases must be distinguished. In the first case, assume that the correct window start time interval of \( b_1 \) is reduced to \([b_1^n.\text{scws}, b_1^l.\text{lcws}] \). Thus

\[
b_1.\text{scws} \leq b_1^n.\text{scws} \leq b_1^l.\text{lcws} \leq b_1.\text{lcws}
\]

holds. This reduction only influences the validity of the fourth term of Property 4.3. Thus, in order to resatisfy \( \text{RelativeTimedCorrectIntervals}(B, RTC) \), only \( b_2.\text{lcwe} \) must be set to \( \text{Min}(b_2.\text{lcwe}, b_1^n.\text{lcws} + b_2.e + lo) \). Then, the
correct window end time interval of $b_3$ remains non-empty when $b_2.scwe \leq \text{Min}(b_2.lcwe, b_3^e.lcw + b_2.e + lo)$ holds. This can be proven as follows:

\[
\begin{align*}
    b_2.scwe & \leq \text{Min}(b_2.lcwe, b_3^e.lcw + b_2.e + lo) \\
    p & \Rightarrow \{\text{Calculus}\} \\
    b_2.scwe & \leq b_2.lcwe \land b_2.scwe \leq b_3^e.lcw + b_2.e + lo \\
    p & \Rightarrow \{\text{Definition 4.1}\} \\
    b_2.scwe & \leq b_3^e.lcw + b_2.e + lo \\
    p & \Rightarrow \{\text{Relation (4.2)}\} \\
    b_2.scwe & \leq b_1.scws + b_2.e + lo \\
    p & \Rightarrow \{\text{Property 4.3}\} \\
    \text{RelativeTimedCorrectIntervals}(B, \text{RTC}) & \text{ before reduction} \\
    & \Rightarrow \{\text{Theorem assumption}\} \\
    \text{true}. 
\end{align*}
\]

In the other three cases, assume that another correct interval of $b_1$ or $b_2$ is reduced. Then, similar reasoning as in the first case can be used to show that all correct intervals of $b_1$ and $b_2$ remain non-empty when $\text{RelativeTimedCorrectIntervals}(B, \text{RTC})$ is resatisfied.

Now, since the correct interval adjustment procedure is called after each assignment of a window to a bead, the algorithm in Figure 4.1 has a polynomial time-complexity $O(|B| \times |\text{RTC}| \times (|B| + |\text{RTC}|))$.

### 4.3 A Window Assignment Algorithm

The algorithm described in the previous section is extended to an algorithm that can find a presumably feasible window assignment (see Figure 4.3). Two extensions are made to incorporate a set of consistency constraints $CC$ and the feasibility conditions given in Properties 2.14 and 2.15.

First, a set of consistency constraints $CC$ is incorporated. Thereby, the algorithm is extended such that it can find a correct window assignment $WA$ for a set of beads with absolute timing constraints, a set of relative timing constraints $RTC$ and a set of consistency constraints $CC$. Thus, $\text{CorrectWindowAssignment}(WA, B, RTC, CC)$ must hold. Note that a consistency constraint between a subgraph of beads $(b_1 \leftrightarrow b_2)$ and a subgraph $(b_1 \leftrightarrow b_4)$ is solved by temporally ordering the two subgraphs. This is achieved by inserting a precedence relation either from $b_2$ to $b_3$ or from $b_4$ to
Assign Presumably_Feasible_WINDOWS (in:B, PR, CC; isout:RTC; out:FOUND, WA)
/* WB is the set of beads to which windows are already assigned */
/* Invariant: CorrectWindowAssignment(WA, WB, RTC, 0) */
WB := ∅;
/* Invariant holds */
/* Initialize each correct interval as large as possible */
for each b ∈ B do
  b.early := b.est; b.late := b.d - b.t;
  b.bseg := b.early - b.c; b.binc := b.d;
end for;
Adjust_Correct_Interval_To_Correctness_Constraints(B, RTC, FOUND);
/* FOUND ⇔ all correctness constraints hold */
/* TCC is the set of transformed consistency constraints */
TCC := ∅;
while FOUND ∧ (TCC ≠ CC) do
  Transform_Consistency_Constraint(B, CC, RTC, TCC, FOUND);
  /* FOUND ⇔ a consistency constraint has been transformed */
  if FOUND then
    Adjust_Correct_Interval_To_Correctness_Constraints(B, RTC, FOUND);
    end if;
end while;
/* FOUND ⇔ TCC = CC ∧ (all correctness constraints hold) */
while FOUND ∧ (WB ≠ B) do
  /* Loop invariant: Feasible_Pairs(WA, WB, PR) */
  Select_Bead(B, WB, RTC, bb);
  Assign_Window(bb, B, RTC, WA, WB);
  Adjust_Correct_Interval(B, RTC, PR, WB, WA, FOUND);
  /* all correctness constraints hold and thus invariant holds */
  /* FOUND ⇔ Loop invariant holds */
end while;
/* FOUND ⇔ Feasible_Pairs(WA, B, PR) */
Check_Feasible_Sets(WA, B, PR, FOUND);
/* FOUND ⇔ Presumably_Feasible_Window_Assignment(WA, B, PR, RTC, 0) */
End.

Figure 4.3: An extended algorithm that can find a presumably feasible window assignment for a set of beads B with a set of relative timing constraints RTC and a set of consistency constraints CC.

The problem of finding a correct window assignment with consistency constraints is transformed into the problem of finding a correct window assignment without consistency constraints. As discussed earlier, the latter problem can be solved easily. Therefore, before windows are assigned to beads, the algorithm shown in Figure 4.3 transforms each consistency constraint in CC into a precedence relation that is added to RTC. After each consistency constraint transfor-
4.3. A Window Assignment Algorithm

In some cases, the correct intervals have to be adjusted subsequently to satisfy the correctness constraints.

Second, the feasibility conditions given in Properties 2.14 and 2.15 are incorporated. Thereby, the algorithm is extended such that it can find a presumably feasible window assignment as well. The assignment of a window to a bead has an effect on the windows of other beads at the same processor. In the algorithm of Figure 4.1 the effects of an assignment of a window to a bead are incorporated in the correct intervals of other beads. Therefore, it seems obvious to incorporate the feasibility conditions in the correct interval adjustment. The feasibility condition of Property 2.14 is indeed incorporated during correct interval adjustment in the part of the algorithm in which windows are assigned. Incorporation of the feasibility condition of Property 2.15 in the correct interval adjustment however results in a high time-complexity of the correct interval adjustment procedure. Therefore, this feasibility condition is only checked at the end of the algorithm when windows are assigned to all beads.

4.3.1 Consistency Constraint Transformation

Transformation of a consistency constraint is done by the procedure in Figure 4.4. The consistency constraint transformation procedure first selects a consistency constraint. Then, the procedure checks if a temporal order is already determined for the selected consistency constraint and if not, a temporal order is chosen. Finally, the procedure inserts a precedence relation according to the temporal order chosen and adds the consistency constraint to set TCC. Note that after this procedure, the algorithm of Figure 4.3 adjusts correct intervals to incorporate the effects of the transformation. This is used in the transformation heuristics presented below.

In Section 3.2.1, the maximum execution intervals of the beads are used to decide whether a temporal order for a consistency constraint is already determined. The same idea is expressed here in terms of the correct intervals. The temporal order of a consistency constraint between a subgraph of beads \((b_1 \rightarrow b_2)\) and a subgraph of beads \((b_3 \rightarrow b_4)\) is already determined if

\[
(b_1.lew - b_1.e < b_4.scews + b_4.e) \lor \\
(b_3.lew - b_3.e < b_2.scews + b_2.e)
\]

holds. Note that, if the first term holds, \((b_1 \rightarrow b_2)\) must be temporally ordered before \((b_3 \rightarrow b_4)\). Similarly, if the second term holds, \((b_3 \rightarrow b_4)\) must be temporally ordered before \((b_1 \rightarrow b_2)\). In addition, a temporal
Transform Consistency Constraint (in: \( B, CC \); inout: \( RTC, TCC \); out: FOUND);
/* select a consistency constraint from \( CC \setminus TCC \) */
bestvalue := 0;
for each \( cc = (b_1, b_2, b_3, b_4) \in CC \setminus TCC \) do
  value := \( \min(b_1, \text{leuc} - b_1, e - (b_4, \text{acuwa} + b_4, e), b_2, \text{leuc} - b_2, e - (b_2, \text{acuwa} + b_2, e)) \);
  if (value < bestvalue) then
    bestcc := cc;
    bestvalue := value;
  end if;
end for;
/* bestcc = (b_1, b_2, b_3, b_4) is the selected consistency constraint */
/* check if a temporal order for bestcc is already determined and */
/* if not, select a temporal order for bestcc */
path1 := "there is a path from a bead in \((b_1 \leadsto b_2)\) to a bead in \((b_2 \leadsto b_4)\)";
path2 := "there is a path from a bead in \((b_3 \leadsto b_1)\) to a bead in \((b_1 \leadsto b_2)\)";
if path1 AND path2 then
  FOUND := false;
else FOUND := true;
if (FOUND)
  then "swap \( b_1 \) and \( b_2 \) with \( b_3 \) and \( b_4 \)";
else if NOT(path1)
  then
    valueacu := \( \max(b_2, \text{acuwa}, b_3, \text{acuwa} + b_3, \text{e})\);\( b_1, \text{leuc} - b_2, \text{e} \);
    valueof := \( \max(b_1, \text{leuc}, b_4, \text{acuwa} + b_4, \text{e})\);\( b_1, \text{leuc} - b_4, \text{e} \);
    \( \text{valueof} < \text{valueacu} \)
    then "swap \( b_1 \) and \( b_2 \) with \( b_3 \) and \( b_4 \)";
  end if;
end if;
end if;
/* FOUND \( \Rightarrow (b_1 \leadsto b_2) \) is temporally ordered before \( (b_2 \leadsto b_4) \) */
/* insert a precedence relation from \( b_2 \) to \( b_3 \) to \( RTC \) */
if FOUND then
  \( RTC := RTC \cup \{(b_2, b_2, e, \infty, b_3)\} \);
  \( TCC := TCC \cup \{ b_3 \} \);
end if;
End.

Figure 4.4: The procedure that transforms a consistency constraint from \( CC \setminus TCC \) to a precedence relation that is added to \( RTC \).
order is also determined if there is a path of precedence relations or relative timing constraints from a bead in subgraph \((b_1 \triangleright b_2)\) to a bead in subgraph \((b_3 \triangleright b_4)\). Note that there may be a precedence relation between beads in subgraph \((b_1 \triangleright b_2)\) and beads in subgraph \((b_3 \triangleright b_4)\) due to previously transformed consistency constraints.

Obviously, it is optimal to first select a consistency constraint for which a temporal order is already determined, because all subsequent adjustments to correct intervals are inevitable. If there is no such consistency constraint, it seems best to select a consistency constraint for which a temporal order is "almost" determined. Note that a temporal order for a consistency constraint between a subgraph of beads \((b_1 \triangleright b_2)\) and a subgraph of beads \((b_3 \triangleright b_4)\) is already determined if formula 4.3 holds. Thus, it seems that the smaller the difference between \(b_4.scws + b_4.e\) and \(b_1.lewe - b_1.e\) or the smaller the difference between \(b_2.scws + b_2.e\) and \(b_3.lewe - b_3.e\), the "more a temporal order is determined". This is used in the consistency constraint selection heuristic. This heuristic selects an untransformed consistency constraint between a subgraph of beads \((b_1 \triangleright b_2)\) and a subgraph of beads \((b_3 \triangleright b_4)\) for which the following is minimal among all untransformed consistency constraints:

\[
\begin{align*}
\text{Min}(b_1.lewe - b_1.e - (b_4.scws + b_4.e),) \\
\text{b_3.lewe - b_3.e - (b_2.scws + b_2.e)).}
\end{align*}
\]

When a consistency constraint is selected, the procedure checks whether a temporal order is already determined. If not, a temporal order is chosen. This is done by a heuristic that attempts to minimize the amount of subsequent adjustment to correct intervals. Suppose subgraph \((b_1 \triangleright b_2)\) is temporally ordered before subgraph \((b_3 \triangleright b_4)\) and a precedence relation is inserted from \(b_2\) to \(b_3\). Then, during the subsequent correct interval adjustment, \(b_3.scws\) is set \(\text{Max}(b_3.scws, b_2.scws + b_2.e)\) and \(b_2.lewe\) is set to \(\text{Min}(b_2.lewe, b_3.lewe - b_3.e)\) to satisfy Property 4.1 for beads \(b_2\) and \(b_3\). Therefore, the amount of adjustment to \(b_3.scws\) and \(b_2.lewe\) is

\[
\begin{align*}
\text{(Max}(b_3.scws, b_2.scws + b_2.e) - b_3.scws) + \\
(b_2.lewe - \text{Min}(b_2.lewe, b_3.lewe - b_3.e)).
\end{align*}
\]

These adjustments are the main adjustments due to the insertion of the precedence relation. Therefore, adjustments to correct intervals of other beads are not considered in the heuristic. When \(b_3 \triangleright b_4\) is temporally ordered before \((b_1 \triangleright b_2)\), there is a similar amount of adjustment to \(b_1.scws\).
and \( b_4, \text{lew} \). Now, the heuristic chooses the temporal order for which this amount of adjustment is minimal.

The selection of a consistency constraint can be done by checking every consistency constraint in \( CC \) once. Therefore, this part of the procedure has a time-complexity \( O(|CC|) \). Checking whether there is a path from one subgraph to another subgraph can be done by a standard procedure that passes through the execution DAG once. Therefore, the part of the procedure that checks whether a temporal order is already determined has a time-complexity \( O(|RTC|) \). Thus, the entire consistency constraint transformation procedure has a time-complexity \( O(|CC| + |RTC|) \).

### 4.3.2 Bead Selection

During the window assignment loop of the algorithm in Figure 4.3, first a bead is selected. Then, a window is assigned to the bead and the correct intervals of all beads are adjusted. The selection of a bead is done in the procedure in Figure 4.5. This procedure selects a bead to which a window

```plaintext
Select_Bead\( (in \ B, WB, RTC; out: bb) \):
/* select a bead from \( B \setminus WB \) */
bestvalue := \( \infty \);
for each \( b \in B \setminus WB \) do
    /* \( \text{rtcs} \) is the number of relative timing constraints of \( b \) */
    b.rsf := \( \infty \); rtcs := 0;
    for each \( (b_1, so, lo, b_2) \in RTC \) do
        if \( b \in \{b_1, b_2\} \) then
            b.rsf := Min(b.rsf, lo - so);
            rtcs := rtcs + 1;
        end if;
    end for;
    if \( \text{rtcs} > 0 \) then
        b.rsf := b.rsf/rtcs;
        b.af := Min(b.lcw - b.acw, b.lcw - b.acw);
        b.f := Min(b.rsf, b.af);
        if \( b.f < \text{bestvalue} \) then
            bb := b;
            bestvalue := b.f;
        end if;
    end if;
end for;
/* \( bb \) is the selected bead */
End.
```

![Figure 4.5: The procedure that selects a bead bb from B \( \setminus \) WB.](image)

is not yet assigned using a heuristic that is based on two observations. The first observation is that a window assignment to a bead with many or with
4.3. A Window Assignment Algorithm

tight relative timing constraints has a big influence on correct intervals of other beads. The second observation is that a window assignment to a bead with small correct intervals results in smaller adjustments of other correct intervals.

The first observation is quantified by a “relative freedom” \( b.rf \) of a bead \( b \). The relative freedom is defined as the smallest value of \( lo - so \) for all relative timing constraints \((so, lo)\) in which \( b \) is involved divided by the number of relative timing constraints of \( b \)

\[
b.rf = \min((b_1, so, lo, b_2) \in RTC : b \in \{b_1, b_2\} : lo - so) / |\{(b_1, so, lo, b_2) \in RTC : b \in \{b_1, b_2\}\}|
\]

Note that this relative laxity must be calculated only once before windows are assigned. In order to express the second observation, the absolute freedom \( b.af \) of a bead \( b \) is introduced. The absolute freedom is defined as the minimum of the lengths of the current correct intervals of bead \( b \)

\[
b.af = \min(b.lcw - b.scw, b.lcw - b.sce)
\]

The two heuristics are combined by defining the freedom \( b.f \) of a bead \( b \) as the minimum of the relative and absolute freedom of \( b \)

\[
b.f = \min(b.rf, b.af)
\]

Finally, the bead selection heuristic selects a bead to which a window is not yet assigned that has minimum freedom. Note that if a window start or end time of a bead is fixed, because the corresponding correct interval contains only one time point, this bead is selected first.

The bead selection procedure selects a bead by calculating the freedom of each bead, while the bead with the minimum freedom is maintained. Calculation of the freedom per bead has a time-complexity \( \mathcal{O}(|RTC|) \). Therefore, the entire procedure has a time-complexity \( \mathcal{O}(|B| \times |RTC|) \).

4.3.3 Window Assignment

When a bead is selected, the algorithm assigns a window to the bead using the procedure in Figure 4.6. This procedure selects a window start time and a window end time from the correct intervals of the bead using a heuristic described below. This is done such that the width of the window is correct
Assign Window(inbb, H, HFC; inewt/WA, WB);
/* calculate the set of contending heads for each bead */
for each bb \in H do
    c.B := \emptyset;
    for each \( b \in AB(b,pr) \setminus \{ b \} \) do
        if \((b,scue) \leq (b,scue) \land b,lcue \leq b,lcue) \)
        then \( c.B := c.B \cup \{ b \} \);
    end for;
end for;
/* find critical path */
bestvalue := \(-\infty\);
for each \( b_1 \rightarrow b_n \rightarrow b_m \in c.B \) do
    execute := \( \sum_{b \in c.B} \sum_{1 \leq i \leq n} \{ (b_\ast, ub, vb) \} \) b,c;
    critfunc := execute/(b_m,lcue - b_1,scue);
    if (critfunc > bestvalue) then
        cp := \( b_1 \rightarrow b_m \);
        bestvalue := critfunc;
    end if;
end for;
/* cp = \( b_1 \rightarrow b_m \) is critical path and \( b = bb \) */
cetl := \( \sum_{b \in c.B} \sum_{1 \leq i \leq n} \{ (b_\ast, ub, vb) \} \) b,c;
ctbl := \( \sum_{b \in c.B} \sum_{1 \leq i \leq n} \{ (b_\ast, ub, vb) \} \) b,c;
ctbr := \( \sum_{b \in c.B} \sum_{1 \leq i \leq n} \{ (b_\ast, ub, vb) \} \) b,c;
/* calculate window width and make it correct */
width := \( \text{Min}(\text{width}, \text{Min}(bb,lcue - bb,scue, bb,scue)) \);
width := \( \text{Max}(\text{width}, \text{Max}(bb,lcue - bb,scue, bb,scue)) \);
/* calculate window for bb in correct intervals */
WA(bb).ws := \( \text{Max}(\text{width}, \text{Max}(bb,scue - bb,scue, bb,scue)) \); 
WA(bb).ws := \( \text{Min}(\text{width}, \text{Max}(bb,scue - bb,scue, bb,scue)) \); 
/* adjust correct intervals of bb and add bb to WB */
bb,scue := \( WA(bb).ws \); 
bb,lcue := \( WA(bb).ws \); 
bb,scue := \( WA(bb).ws \); 
bb,lcue := \( WA(bb).ws \); 
W := W \cup \{ bb \}.

End.

Figure 4.6: The procedure that assigns a window to bead bb, adjusts the correct intervals of bb accordingly and adds bb to WB.
as defined in the text after Definition 4.2. Finally, the procedure adjusts the correct intervals of the bead accordingly and adds the bead to set \( WB \).

The heuristic that assigns a window to a bead \( b \) is based on a "critical path" through the execution DAG that restricts the window of \( b \) the most. A path \( b_1 \rightarrow \ldots \rightarrow b_n \) is a sequence of \( n \) beads connected by precedence relations and relative timing constraints. A critical function \( critfunc \) for a path expresses that the path is more critical when the total execution time on the path is larger and the length of the interval in which the beads on the path have to be executed is smaller

\[
critfunc(b_1 \rightarrow \ldots \rightarrow b_n) = \frac{execute(b_1 \rightarrow \ldots \rightarrow b_n)}{length(b_1 \rightarrow \ldots \rightarrow b_n)}.
\]

The length of a path is simply defined as the difference between the \( lcwe \) of the last block on the path and the \( scws \) of the first block on the path

\[
length(b_1 \rightarrow \ldots \rightarrow b_n) = b_n.lcwe - b_1.scws.
\]

In order to define the execution time of a path, it is important to note that the execution of a bead depends on the execution of other beads that contend for the same processor. A bead \( b \) is defined to contend with a different bead \( b' \) on the path if \( b \) is at the same processor as \( b' \) and if the interval \([b.scws, b.lcwe]\) is completely contained in the interval \([b'.scws, b'.lcwe]\). The set \( CB \) of beads contending for the beads on a path is defined as

\[
CB(b_1 \rightarrow \ldots \rightarrow b_n) = \{ b \in B \setminus \{b_1, \ldots, b_n\} : \exists i : b \in AB(b_i, pr) : b_i.scws \leq b.scws \land b.lcwe \leq b_i.lcwe \}\).
\]

Then, the total execution time of a path is defined as the sum of the execution times of all beads on and all contending beads for the path

\[
execute(b_1 \rightarrow \ldots \rightarrow b_n) = \sum_{b \in \{b_1, \ldots, b_n\} \cup CB(b_1 \rightarrow \ldots \rightarrow b_n)} b.e.
\]

Finally, a path through a bead \( b \) is a critical path for \( b \) if it has the largest critical function among all paths through \( b \). Once a critical path \( b_1 \rightarrow \ldots \rightarrow b_n \) where \( b = b_k \) has been found, a window is assigned such that the execution
time of the path is divided evenly over the path. Therefore, first a window
width $\text{width}$ is selected such that

$$\text{width}$$

$$= \frac{\text{execute}(b_k \rightarrow b_k)}{\text{execute}(b_1 \rightarrow b_n)} \times (b_n.lcwe - b_1.scws).$$

Once the width is calculated, it is adjusted by the minimum amount to make it
correct as defined in the text after Definition 4.2. Recall that the width
of a window of a bead $b$ is correct if it is at least the execution time of the
bead and at most the largest window width $b.lw$ given by Definition 4.2.
Furthermore, the width is adjusted such that it is at least the minimum width
based on the correct intervals of $b$. Then, a window start time $WA(b).ws$ is
selected such that

$$WA(b).ws$$

$$= b_1.scws \times \text{execute}(b_{k+1} \rightarrow b_n)/$$

$$\text{execute}(b_1 \rightarrow b_{k-1}) + \text{execute}(b_{k+1} \rightarrow b_n))$$

$$+ (b_n.lcwe - width) \times \text{execute}(b_1 \rightarrow b_{k-1})/$$

$$\text{execute}(b_1 \rightarrow b_{k-1}) + \text{execute}(b_{k+1} \rightarrow b_n))_.$$

Once a window start time is calculated, it is adjusted by the minimum
amount such that it is an element of the correct window start time interval.
Finally, the window end time $WA(b).we$ is set to the selected window start
time plus the width of the window. Note that these adjustments are always
possible, because the correct intervals of bead $b$ satisfy the correctness
constraints.

The time-complexity of the window assignment procedure is dominated
by the part that searches for a critical path. The critical path is found by
checking the path between each pair of beads. Since a path contains at most
all precedence relations and relative timing constraints, the entire procedure
has a time-complexity $O(|RTC| \times |B|^2)$.

### 4.3.4 Correct Interval Adjustment

The feasibility condition of Property 2.14 is incorporated in the correct
interval adjustment procedure given in Figure 4.7 during the part of the algo-
rithm in which windows are assigned. Therefore, in addition to the invariant
of the algorithm, the invariant of the loop in which windows are assigned is \( \text{FeasiblePairs}(WA, WB, PR) \). Initially \( WB = \emptyset \) and thus the loop invariant is satisfied. Then, when a window is assigned to each bead, i.e., \( WB = B \), \( \text{FeasiblePairs}(WA, B, PR) \) holds and the feasibility condition of Property 2.14 is satisfied. Note that it might not always be possible to satisfy the loop invariant after assignment of a window.

Property 2.14 expresses that for each pair of beads at the same processor, the sum of the execution times of the beads must be at most the length of the interval between the window start time of one of the beads and the window end time of the other bead. Obviously, this must also hold for the smallest correct window start times and the largest correct window end times. This is expressed by the following property.

**Property 4.5** Let \( B \) be a set of beads assigned to a set of processors \( PR \). Let \( WA \) be a window assignment to the beads in \( B \). Then, the feasible correct interval pairs property for \( WA, B \) and \( PR \) is defined as

\[
\text{FeasibleCorrectIntervalPairs}(WA, B, PR) \\
\Leftrightarrow \\
(\forall pr \in PR : (\forall b_1, b_2 \in AB(pr) : b_1 \neq b_2 : \\
\quad b_1.e + b_2.e \leq b_2.lewe - b_1.scws \lor \\
\quad b_1.e + b_2.e \leq b_1.lewe - b_2.scws \\
)).
\]

Property 4.5 is called an interval feasibility constraint. Now, it is shown that when this interval feasibility constraint and Property 4.4 are satisfied, the loop invariant \( \text{FeasiblePairs}(WA, WB, PR) \) of the algorithm of Figure 4.3 holds. For this purpose, Property 4.4 is combined with Property 4.5. Thereby, the latter property is transformed into Property 2.14, where \( B \) is substituted by \( WB \).

In the algorithm of Figure 4.1, correct intervals are adjusted to the correctness constraints. Now, in the window assignment loop of the extended algorithm in Figure 4.3, correct intervals are adjusted to the feasibility constraint as well. This is done by the procedure in Figure 4.7. This procedure repeatedly adjusts the correct intervals to the correctness constraints and the interval feasibility constraint. Adjustment to the correctness constraints is done by the procedure given in Figure 4.2. Adjustment to the interval
Adjust_Correct_Intervals (in:B, RTC, PR, WB; inout:WA; out:FOUND);
FOUND := false;
/* na is the number of iterations of the adjustment loop */
while ~FOUND ( A na < MNI) do
  Adjust_Correct_Intervals_To_Correctness_Constraints(B, RTC, EXISTS);
  /* All correctness constraints hold */
  if (na < MNI) then
    C := B; nc := 0;
    /* C is the set of beads of which a correct interval might have changed */
    /* nc is the number of iterations of the checking loop */
    while (C ≠ ∅ ∧ nc < MNC) do
      h1 := "a random element from C";
      C := C \ {h1};
      for each b2 ∈ AB(b1, pr) \ {h1} do
        Satisfy_Interval_Feasibility_Constraint (WB, b1, b2, WA, C);
      end for;
    end while;
    nc := nc + 1;
  end if;
  na := na + 1;
end while;
/* FOUND */ Feasible_Pairs(WA, WB, PR) */
End.

Figure 4.7: The procedure that adjusts the correct intervals of a set of beads B assigned to a set of processors PR to the correctness constraints and the interval feasibility constraint given a set of relative timing constraints RTC.

Feasibility constraint is described below. As will become clear, adjustment to the interval feasibility constraint influences the validity of the correctness constraints and vice versa. Due to this mutual influence, it is unknown whether and when the procedure ends. Therefore, the number of iterations of the adjustment loop is bounded by a maximum MNI. When the procedure has made MNI iterations, it terminates such that the correct intervals satisfy the correctness constraints but not the interval feasibility constraint.

The correct intervals are adjusted to the interval feasibility constraint by repeatedly checking a bead b1 of which a correct interval has changed. This is done by checking the interval feasibility constraint for b1 and each other bead b2 at the same processor in the procedure in Figure 4.8. However, it is not always possible to satisfy the interval feasibility constraint for two beads b1 and b2. Thus, it is unknown whether and when the interval feasibility constraint will be completely satisfied. Therefore, the number of iterations
4.3. A Window Assignment Algorithm

Satisfy Interval Feasibility Constraint (in:WB; inout:b1,b2,WA,C):
ordered := false;
if (b1.e + b2.e ≤ b2.owe - b1.acwa) ∧ (b1.e + b2.e ≤ b1.owe - b2.acwa)
then ordered := true
else if (b1.e + b2.e > b2.owe - b1.acwa)
then "swap b1 and b2"; ordered := true;
else order := "a random element from {true,false}";
if ~(order) then "swap b1 and b2";
/* b1 is temporally ordered before b2 */
b2.owe := b1.acwa + b1.e + b2.e, thus increase b2.owe */
b2.owe := b1.acwa + b1.e + b2.e;
/* assure Property 4.2 for b2 */
b2.owe := Min(b2.owe, b2.d);  
if (b1.e + b2.e > b2.owe - b1.acwa) then
/* b1.acwa > b2.owe - b1.e - b2.e, thus decrease b1.acwa */
b1.acwa := b2.owe - b1.e - b2.e;  
/* assure Property 4.2 for b2 */
b1.acwa := Max(b1.acwa, b1.owe);
end if;
if (b1.e + b2.e ≤ b2.owe - b1.acwa)
then ordered := true;
end if;
/* ordered ⇔ b1 is temporally ordered before b2 */
if ordered then
/* satisfy (4.4): b1.acwa + b1.e ≤ b2.acwa ∧ b1.owe ≤ b2.owe - b2.e */
b2.acwa := Max(b2.acwa, b1.acwa + b1.e);
b1.owe := Min(b1.owe, b2.owe - b2.e);
for each b ∈ {b1, b2} do
/* satisfy Definition 4.1 */
b.owe := Max(b.owe, b.acwa);
b.acwa := Min(b.acwa, b.owe);
/* satisfy Property 4.1 */
b.acwa := Max(b.acwa, b.acwa + b.e);
b.owe := Min(b.owe, b.owe - b.e);
if b ∈ WH then
/* satisfy Property 4.4 */
b.acwa := b.acwa;
WA(b).ow := b.owe;
b.acwa := b.owe;
WA(b).we := b.owe;
end if;
end for;
C := C ∪ \{b1, b2\};
end if;
End.

Figure 4.8: The procedure that adjusts the correct intervals of two beads b1 and b2 to satisfy the interval feasibility constraint and formula (4.4).
in which a bead \( b_1 \) is checked is bounded by a maximum \( MNC \).

In the procedure, the correct intervals are only adjusted in the following two cases. First, the constraint is satisfied for beads \( b_1 \) and \( b_2 \) because only one of the two terms of Property 4.5 is satisfied. In this case, the correct intervals determine a temporal order among beads \( b_1 \) and \( b_2 \). Suppose, \( b_1 \) must be temporally ordered before \( b_2 \). Then, in order to make the correct intervals more accurate, they are adjusted according to the temporal order. Therefore, \( b_2.sews \) is increased and \( b_1.lcwe \) is decreased such that the following relation holds

\[
b_1.sews + b_1.e \leq b_2.sews \land b_1.lcwe \leq b_2.lcwe - b_2.e.
\]  

(4.4)

In the second case, the interval feasibility constraint is not satisfied for \( b_1 \) and \( b_2 \). In this case, the two beads can not be temporally ordered based on the correct intervals. Therefore, a temporal order is randomly selected. Suppose \( b_1 \) is temporally ordered before \( b_2 \). Then, it is tried to satisfy the first term of the interval feasibility constraint. Therefore, \( b_2.lcwe \) is increased without violating Property 4.2. If the first term does not hold after this adjustment, \( b_1.sews \) is decreased without violating Property 4.2. Then, only if the first term holds, \( b_2.sews \) is increased and \( b_1.lcwe \) is decreased such that (4.4) holds. Thereby, the correct intervals are adjusted to reflect the temporal order. Note that this is always possible when the first term holds.

When a correct interval of a bead is changed, both correct intervals of the bead are immediately adjusted to satisfy Definition 4.1, Property 4.1 and Property 4.4. Note that windows that are already assigned can be shifted during adjustment to the interval feasibility constraint. Since Property 4.2 is not violated during the adjustments, all correctness constraints except Property 4.3 are satisfied after adjustment to the interval feasibility constraint in the inner while-loop of the correct interval adjustment procedure in Figure 4.7. Therefore, the procedure that adjusts the correct intervals to the correctness constraints in the outer while-loop of the procedure in Figure 4.7 can be called.

Since at most each pair of beads is checked for the interval feasibility constraint, the part of the procedure in Figure 4.7 that adjusts correct intervals to the interval feasibility constraint has a time-complexity \( O(|B|^2) \). As mentioned earlier, the procedure that adjusts the correct intervals to the correctness constraints has a time-complexity \( O(|RTC| \times (|B| + |RTC|)) \). Therefore, the entire correct interval adjustment procedure has a time-complexity \( O(|RTC| \times (|B| + |RTC|) + |B|^2) \).
4.3. A Window Assignment Algorithm

4.3.5 Feasibility Condition Check

The feasibility condition of Property 2.15 is only checked at the end of the algorithm when windows are assigned to all beads. This is done by the procedure in Figure 4.9. The procedure first sorts the beads by non-decreasing window end time. This is done by a standard sorting procedure that has a time-complexity less than $O(|B|^2)$. Then, for each window start time, the procedure passes forwards through the sorted list of beads only once, while maintaining and checking the execution sum. Therefore, the entire procedure has a time-complexity $O(|B|^2)$.

The time-complexity of the initialization part of the extended window assignment algorithm of Figure 4.3 is dominated by the correct interval adjustment procedure and thus equals $O(|RTC| \times (|B| + |RTC|))$. The consistency constraint transformation part of the algorithm calls the consistency constraint transformation procedure and the correct interval adjustment procedure for each consistency constraint. Therefore, this part of the algorithm has a time-complexity $O(|CC| \times (|CC| + |RTC| \times (|B| + |RTC|)))$. The window assignment part of the algorithm calls the bead selection procedure, the window assignment procedure and the extended correct interval
adjustment procedure for each bead. Therefore, this part of the algorithm has a time-complexity \( O(|B| \times |RTC| \times (|B|^2 + |RTC|)) \). Finally, the algorithm checks whether the feasibility condition of Property 2.15 holds. This part of the algorithm has a time-complexity \( O(|B|^2) \). Summarizing, the entire extended window assignment algorithm has a time-complexity \( O(|CC| \times (|CC| + |RTC| \times (|B| + |RTC|)) + |B| \times |RTC| \times (|B|^2 + |RTC|)) \).

Results of performance measurements are presented in Chapter 6.

4.4 Conclusion

In this chapter, a heuristic constructive algorithm is presented that can solve the window assignment problem of the third step of the approach. For this purpose, consistency constraints are transformed into relative timing constraints before windows are assigned. Instead of backtracking to undo certain window assignment decisions, the algorithm shifts already assigned windows in order to satisfy the feasibility condition of Property 2.14. Performance measurements of the algorithm can be found in Chapter 6. The results show that this is a powerful technique to solve the problem.
Chapter 5

Local Scheduling

In this chapter, a local scheduling algorithm is presented that can solve the local scheduling problem of the third step of the approach. The local scheduling algorithm constructs a local schedule by repeatedly adding beads to the end of a partial feasible local schedule. Bead selection is done using an enhanced version of the earliest deadline first strategy. This is based on three theorems that describe certain characteristics of the problem. If the partial feasible local schedule can not be feasibly extended with a new bead, the algorithm uses intelligent backtracking heuristics. Thereby, certain beads at the end of the partial feasible local schedule are removed and replaced by different beads. Results of performance measurements presented in Chapter 6, show that the algorithm finds a feasible local schedule in almost all cases. In addition, the results show that on the average the computation time of the algorithm is quadratic in the number of beads.

5.1 The Local Scheduling Problem

The local scheduling problem concerns a set of beads $P_B$ that have to scheduled at the same processor. Each bead $b \in P_B$ has a window $(b.w_s, b.w_e)$ in which it has to be scheduled. As defined in Section 2.6.3, the local scheduling problem is to find a feasible local schedule $LS(P_B)$ for the set of beads $P_B$ at the same processor. Thus, $FeasibleLocalSchedule(LS(P_B))$ holds. As discussed in Section 2.6.3, the local scheduling problem is NP-complete. The
problem is equivalent to the problem of finding a single processor schedule for a set of non-preemptable tasks with release times and deadlines. The tasks to be scheduled are the heads, each window start is a release time and each window end is a deadline.

The current literature covers several approaches and algorithms to solve this problem. In [21], a special instance of the problem is considered. In this instance, the deadline of any task \( t_i \) is at most the deadline of any other task \( t_j \), if the release time of \( t_i \) is at most the release time of \( t_j \). Therefore, no window is nested in any other window. For this special case, it is shown that the earliest deadline first strategy can find a feasible local schedule whenever one exists. Consequently, for this case, an algorithm with polynomial time-complexity. However, the NP-completeness of the problem implies that in the general case only algorithms with exponential time-complexity can find a feasible local schedule whenever one exists. Since the number of local schedules equals \( |P|! \), the search space of a reasonably sized problem instance is tremendously large. Various techniques have been developed to reduce the search space and thus improve average performance.

Heuristic techniques are often used to quickly find a solution by significantly reducing the search space. With these techniques, a schedule is usually constructed by repeatedly adding a task to an already constructed partial schedule without backtracking. In each step, an evaluation function determines which task should be added. Already decades ago, several heuristic functions have been proposed. In [19], the earliest deadline first strategy is introduced. With this strategy, the task selected next is the task with the earliest deadline. In [36], the least laxity first strategy is proposed. The laxity of a task is the difference between the release time of the task and deadline of the task minus the execution time of the task. In [24], the earliest release time first strategy is presented. Since there is no backtracking in these algorithms, the scheduling cost is relatively low. On the other hand, the use of such straightforward heuristic functions usually results in a small percentage of feasible schedules found. Therefore, more recently, these heuristic functions have been enhanced to be used for run-time scheduling [26, 27, 28, 54]. However, the percentage of feasible schedules found with these enhanced heuristics is still too small in the context of a pre-run-time scheduling approach.

Branch-and-bound techniques are often used to enumerate possible solutions. With such a technique, the search space is represented by a search tree in which the root at level 0 contains the empty schedule, while each node at level \( k \) corresponds to a partial schedule containing \( k \) tasks. The children of
5.1. *The Local Scheduling Problem*

A node corresponds to extensions of the partial schedule of that node with a single task. Thus, leaves of the tree are complete schedules. A bound is associated with each node which reflects the possibility that the partial schedule of the node can be extended to a feasible schedule. The search process starts at the root node and branches through the tree guided by heuristics until a leaf node with a feasible or optimal schedule is reached. Parts of the tree are pruned only when the bound in a node indicates that a feasible schedule can definitely not be reached via this node. When all the children of a node are pruned, backtracking to the father of the node is necessary. Therefore, a branch-and-bound algorithm is usually optimal in the sense that it can find a solution whenever one exists. Hence, all possible solutions might be enumerated when the algorithm cannot prune any part of the tree. Consequently, a branch-and-bound algorithm has an exponential time-complexity which in the general case leads to a relatively high scheduling cost.

In [5], a branch-and-bound algorithm is proposed, in which the bound of a node depends on the feasibility of all the extensions in the children of the node. The algorithm also utilizes situations in which a partial schedule finishes prior to the release times of all remaining tasks. In [2], a lower bound on the maximum tardiness is calculated for a partial schedule by extending the schedule using the preemptive earliest deadline first strategy. The tardiness of a task in a schedule is the maximum of the completion time of the task minus the deadline of the task and 0. A different enumeration technique is used in [29] to minimize the maximum lateness. The lateness of a task in a schedule is the completion time of the task minus the deadline of the task. An initial, possibly infeasible, schedule is first computed using straightforward heuristics, and successively refined to construct a feasible schedule. A similar technique is used in [8] to solve a problem that is shown to be equivalent to the problem of finding a single processor schedule for a set of non-preemptible tasks with release times and deadlines [6].

Besides heuristic and branch-and-bound approaches, a decomposition technique has been proposed recently to solve the latter problem [53]. With this technique, a set of tasks is decomposed into a sequence of subsets such that the scheduling of a subset is independent of the scheduling of all other subsets. The overall scheduling cost is equal to the sum of the scheduling costs for each subset plus the cost for decomposition. This technique is based on an analytical approach to the problem presented in [13]. In the analysis, particular properties of sequences are defined by examining the release times and deadlines of tasks. This significantly reduces the number of sequences to be considered in solving the problem. Although this technique gives good
results, it strongly depends on the possibility to decompose the set of tasks into subsets that are small enough to be sequenced in reasonable time.

In this chapter, a local scheduling algorithm is presented that combines the advantages of both heuristic and branch-and-bound approaches. The algorithm incorporates an enhanced version of the earliest deadline first heuristic and intelligent backtracking heuristics [46]. An important difference with the backtracking mechanism of the conventional branch-and-bound technique is that parts of the tree can be pruned although they might lead to a feasible local schedule. In order to present the heuristic function and the backtracking mechanism, the problem is first analyzed in detail.

5.2 Problem Analysis

One of the similarities between most heuristic and branch-and-bound approaches is that a local schedule is constructed by repeatedly adding a bead to the end of an already constructed partial local schedule. This characteristic is adopted in the approach presented here and the local scheduling problem can be analyzed in this context. Therefore, the concept of concatenating two local schedules is defined first.

**Definition 5.1 (Concatenate operator)** Let \( LS(\mathcal{PB}_1) = (b^1_1, \ldots, b^1_{\mathcal{PB}_1}) \) be a local schedule for the set of beads \( \mathcal{PB}_1 \) and let \( LS(\mathcal{PB}_2) = (b^2_1, \ldots, b^2_{\mathcal{PB}_2}) \) be a local schedule for a different set of beads \( \mathcal{PB}_2 \). The concatenate operator \( \oplus \) concatenates two local schedules to a new local schedule for the set of beads \( \mathcal{PB}_1 \cup \mathcal{PB}_2 \)

\[
LS(\mathcal{PB}_1) \oplus LS(\mathcal{PB}_2) = (b^1_1, \ldots, b^1_{\mathcal{PB}_1}, b^2_1, \ldots, b^2_{\mathcal{PB}_2}).
\]

The analysis considers a set of scheduled beads \( SB \subseteq \mathcal{PB} \) and a set of unscheduled beads \( UB = \mathcal{PB} \setminus SB \). In addition, \( LS(SB) = \langle b_1, \ldots, b_{|SB|} \rangle \) is the partial feasible local schedule already constructed for the beads in \( SB \).

Three theorems are derived and proven which identify sequences of unscheduled beads that preserve the feasibility when added to the partial feasible local schedule \( LS(SB) \). This means that if \( LS(SB) \) is a prefix of a feasible local schedule, then \( LS(SB) \) extended with such a sequence of beads is also a prefix of a feasible local schedule. Such a sequence of beads is called a feasibility preserving extension for the partial feasible local schedule. Therefore, the concept of a prefix of a feasible local schedule is defined as follows.
Property 5.1 Let $SB$ be the set of scheduled beads and $UB$ the set of
unscheduled beads. Let $LS(SB)$ be a partial feasible local schedule for $SB$. Then, the prefix feasible local schedule property for $LS(SB)$ is defined as

$$PrefixFeasibleLocalSchedule(LS(SB))$$

$$\Rightarrow$$

$$\exists LS(UB): \text{Feasible}(LS(SB) \oplus LS(UB)) \}.$$ 

In Section 5.3, a local scheduling algorithm is presented. The theorems
are applied in this algorithm after a bead is added to the end of the par-
tial local schedule and. The effects of the theorems is incorporated in the
backtracking heuristic of the algorithm. Thereby, the set of possible local
schedules to be considered is reduced significantly.

5.2.1 Feasibility Preserving Beads

The first theorem identifies a feasibility preserving extension that consists of
a single unscheduled bead $b \in UB$. Obviously, it is preferred that, if $LS(SB)$
is a prefix of a feasible local schedule, then $LS(SB) \oplus (b)$ is also a prefix of
a feasible local schedule. Such a bead is called a feasibility preserving bead
for $LS(SB)$.

Property 5.2 Let $SB$ be the set of scheduled beads and $UB$ the set of
unscheduled beads. Let $LS(SB)$ be a partial feasible local schedule for $SB$
and let $b \in UB$. Then, the feasibility preserving bead property for $LS(SB)$
and $b$ is defined as

$$FeasibilityPreservingBead(LS(SB), b)$$

$$\Leftrightarrow$$

$$PrefixFeasibleLocalSchedule(LS(SB)) \Rightarrow$$

$$PrefixFeasibleLocalSchedule(LS(SB) \oplus (b)).$$

The first theorem states when an unscheduled bead $eb$ with the smallest
window end time among all unscheduled beads is feasibility preserving for
$LS(SB)$. This is the case when extending $LS(SB)$ with an unscheduled
bead that starts earlier than $eb$ does not lead to a feasible local schedule.
For convenience, the start time of a bead that is added to a partial local
schedule is defined as follows.
Definition 5.2 (Bead start time after local schedule) Let $SB$ be the set of scheduled beads and $UB$ the set of unscheduled beads. Let $LS(SB) = \langle b_1, ..., b_{|SB|} \rangle$ be a partial feasible local schedule for $SB$ and let $b \in UB$. Then, the start time of a bead $b$ after a local schedule $LS(SB)$ is defined as

$$st(b, LS(SB)) = \begin{cases} 
\max (b.ws, st(b_{|SB|} + e_{|SB|})) & \text{if } SB \neq \emptyset \\
 b.ws & \text{otherwise}
\end{cases}$$

Now, the first theorem is formally defined as follows.

Theorem 5.1 Let $SB$ be the set of scheduled beads and $UB$ the set of unscheduled beads. Let $LS(SB) = \langle b_1, ..., b_{|SB|} \rangle$ be a partial feasible local schedule for $SB$. Let $eb \in UB$ such that: $\forall b \in UB \setminus \{eb\} : b.ws \geq eb.ws$.

Then, the following holds:

$$\forall b \in UB \setminus \{eb\} : st(b, LS(SB)) < st(eb, LS(SB)) : \quad \Rightarrow \quad PrefixedFeasibleLocalSchedule(LS(SB) \oplus \langle eb \rangle)$$

FeasibilityPreservingBead(LS(SB), eb).

Proof For the proof of Theorem 5.1, it is assumed that the left hand side of the implication holds. Then, it is shown that an arbitrary feasible local schedule with $LS(SB)$ as a prefix, can always be transformed into a feasible local schedule with $LS(SB) \oplus \langle eb \rangle$ as a prefix. Consequently, block $eb$ is feasibility preserving for $LS(SB)$. The proof presented here is informal and shows intuitively that the theorem holds.

Consider an arbitrary feasible local schedule $LS(PB) = \langle b_1, ..., b_l \rangle$ with $LS(SB)$ as prefix, where $|PB| = l$. Suppose $SB$ contains $sb$ beads. Then, the first $sb$ beads of $LS(PB)$ form $LS(SB)$ and $UB$ consists of beads $b_{sb+1}$ to $b_l$. Suppose $sb+k, 1 \leq k \leq l-sb$, is the index of bead $eb$ in $LS(PB)$. The definition of bead $eb$ can be equivalently formulated in terms of $LS(PB)$ as

$$\forall i : sb+1 \leq i \leq l : wc_i \geq wc_{eb+k}.$$  (5.1)

The feasibility of $LS(PB)$ implies that $LS(SB) \oplus \langle eb \rangle$ is a prefix of a feasible local schedule and thus $PrefixedFeasibleLocalSchedule(LS(SB) \oplus \langle eb \rangle)$ holds. Under the assumption that the left hand side of the implication of Theorem 5.1 holds, this implies that

$$st(b_{sb+1}, LS(SB)) \geq st(b_{sb+k}, LS(SB)).$$
5.2. Problem Analysis

From this relation, it can be derived, using Definition 5.2, that this implies that the window start time of bead $eb$ is at most the start time of bead $b_{sb+1}$ in $LS(PB)$

$$w_{sb+k} \leq st_{sb+1}.$$  \hfill (5.2)

Local schedule $LS(PB)$ can be transformed into a local schedule $LS'(PB) = \langle b'_1, ..., b'_l \rangle$ with $LS(SB) \oplus \{eb\}$ as a prefix. This is done by swapping beads $b_{sb+1}$ to $b_{sb+k-1}$ with bead $b_{sb+k}$. This transformation can be represented by the following mapping between the beads of $LS'(PB)$ and $LS(PB)$ (see Figure 5.1):

$$(\forall i : 1 \leq i \leq sb : b'_i = b_i) \land$$

$$(b'_{sb+1} = b_{sb+k}) \land (\forall i : sb + 2 \leq i \leq sb + k : b'_i = b_{i-1}) \land$$

$$(\forall i : sb + k + 1 \leq i \leq l : b'_i = b_i).$$

It has to be proven that $LS'(PB)$ is feasible, and thus that each bead in

Local Schedule $LS(PB)$

![Diagram of Local Schedule LS(PB)]

Local Schedule $LS'(PB)$

![Diagram of Local Schedule LS'(PB)]

Figure 5.1: Transformation from local schedule $LS(PB)$ into local schedule $LS'(PB)$ in the proof of Theorem 5.1.

$LS'(PB)$ ends before its window end time. This is informally shown in Figure 5.1. First, heads $b'_1$ to $b'_n$ form partial feasible local schedule $LS(SB)$ and thus each of these beads ends before its window end time. Second,
Chapter 5. Local Scheduling

formula (5.2) implies that the latest time at which bead $b'_{sh+1}$ is started is the start time of $b_{sh+1}$. Since $LS(PB)$ is feasible, this bead ends before its window end time. Third, each bead $b'_{sh+2}$ to $b'_{sh+h}$ is shifted forward by the transformation for at most $e_{sh+h}$ time units and thus end before $we'_{sh+h}$. Formula (5.1) and the mapping imply that the use of each bead $b'_{sh+2}$ to $b'_{sh+h}$ is at least $we'_{sh+1}$. Thus, each of these beads ends before its window end time. Finally, each bead $b'_{sh+h}$ to $b'_{f}$ is not scheduled later as in $LS(PB)$, and thus each of these beads ends before its window end time. This completes the proof.

5.2.2 Feasibility Preserving Bead Sequences

The second theorem identifies a feasibility preserving extension that consists of a sequence of $u$ unscheduled beads $(b_1, ..., b_u)$ from $UB$. Obviously, it is preferred that, if $LS(SB)$ is a prefix of a feasible local schedule, then $LS(SB) \oplus (b_1, ..., b_u)$ is also a prefix of a feasible local schedule. Such a sequence of beads is called a feasibility preserving bead sequence for $LS(SB)$.

**Property 5.3** Let $SB$ be the set of scheduled beads and $UB$ the set of unscheduled beads. Let $LS(SB)$ be a partial feasible local schedule for $SB$. Let $b_1$ to $b_u$ be $u$ beads of $UB$. Then, the feasibility preserving bead sequence property for $LS(SB)$ and sequence $(b_1, ..., b_u)$ is defined as

$$FeasibilityPreservingBeadSequence(LS(SB), (b_1, ..., b_u))$$

$$\Leftrightarrow$$

$$PrefixFeasibleLocalSchedule(LS(SB)) \Rightarrow$$

$$PrefixFeasibleLocalSchedule(LS(SB) \oplus (b_1, ..., b_u)).$$

An almost trivial but important consequence of this property is that each bead $b_i$ in $(b_1, ..., b_u)$ is a feasibility preserving bead for $LS(SB) \oplus (b_1, ..., b_{i-1})$. This is expressed by the following corollary.

**Corollary 5.1** Let $SB$ be the set of scheduled beads and $UB$ the set of unscheduled beads. Let $LS(SB)$ be a partial feasible local schedule for $SB$. Let $b_1$ to $b_u$ be $u$ beads of $UB$. Then, the following holds:

$$FeasibilityPreservingBeadSequence(LS(SB), (b_1, ..., b_u))$$

$$\Rightarrow$$
5.2. Problem Analysis

\[(\forall i : 1 \leq i \leq u:\]
\[\text{FeasibilityPreservingBead}(LS(SB) \oplus (b_1, ..., b_{i-1}), b_i)\]
\).  

Before the second theorem is presented, the connected property is defined for a sequence of beads added to a partial local schedule. Two successive beads are connected if the start time of the second bead equals the end time of the first one.

**Property 5.4** Let \(SB\) be the set of scheduled beads and \(UB\) the set of unscheduled beads. Let \(LS(SB)\) be a partial feasible local schedule for \(SB\). Let \(b_1\) to \(b_u\) be \(u\) beads of \(UB\). Then, the connected sequence property for \(LS(SB)\) and sequence \((b_1, ..., b_u)\) is defined as

\[\text{ConnectedSequence}(LS(SB), (ub_1, ..., ub_u))\]
\[\Leftrightarrow\]
\[(\forall i : 2 \leq i \leq u \land b_i \in LS(SB) \oplus (b_1, ..., b_u) : \]
\[b_i.st = b_{i-1}.et + b_{i-1}.e)\]
\).

Let \((eb_1, ..., eb_u)\) be a sequence of \(u\) unscheduled beads with the smallest window end times among all unscheduled beads. Suppose that this sequence is connected when it is added to \(LS(SB)\) and that \(LS(SB) \oplus (eb_1, ..., eb_u)\) is a partial feasible local schedule. The second theorem then states that this sequence is a feasibility preserving bead sequence for \(LS(SB)\) if extending \(LS(SB)\) with an unscheduled bead that starts earlier than \(eb_1\) does not lead to a feasible local schedule.

**Theorem 5.2** Let \(SB\) be the set of scheduled beads and \(UB\) the set of unscheduled beads. Let \(LS(SB)\) be a partial feasible local schedule for \(SB\). Let \(eb_1\) to \(eb_u\) be \(u\) beads of \(UB\) such that

\[\text{ConnectedSequence}(LS(SB), (eb_1, ..., eb_u)) \land \]
\[\text{Feasible}(LS(SB) \oplus (eb_1, ..., eb_u)) \land \]
\[(\forall i : 1 \leq i \leq u : (\forall b \in UB \setminus \{eb_1, ..., eb_u\} : b.et \geq eb_1.et))\].

Then, the following holds:

\[(\forall b \in UB \setminus \{eb_1\} : st(b, LS(SB)) < st(eb_1, LS(SB))):\]
Chapter 5. Local Scheduling

\[ \neg \text{PrefixFeasibleLocalSchedule}(LS(SB) \ominus \langle b \rangle) \]

\[ \Rightarrow \]

\[ \text{FeasibilityPreservingBeadSequence}(LS(SB), \langle eb_1, \ldots, eb_u \rangle). \]

**Proof** For the proof of Theorem 5.2, it is assumed that the left hand side of the implication holds. Then, it is shown that an arbitrary feasible local schedule with \( LS(SB) \) as a prefix, can always be transformed into a feasible local schedule with \( LS(SB) \oplus \langle eb_1, \ldots, eb_u \rangle \) as a prefix. Consequently, sequence \( \langle eb_1, \ldots, eb_u \rangle \) is feasibility preserving for \( LS(SB) \). The proof presented here is informal and shows intuitively that the theorem holds.

Consider an arbitrary feasible local schedule \( LS(PB) = \langle b_1, \ldots, b_l \rangle \) with \( LS(SB) \) as a prefix, where \( |PB| = l \). Suppose \( SB \) contains \( sb \) beads. Then, the first \( sb \) beads of \( LS(PB) \) form \( LS(SB) \) and \( UB \) consists of beads \( b_{sb+1} \) to \( b_l \). Suppose that the beads of sequence \( \langle eb_1, \ldots, eb_u \rangle \) are positioned in \( LS(PB) \) at indices \( k_1 < \ldots < k_u \). In order to relate the indices 1 to \( u \) in \( \langle eb_1, \ldots, eb_u \rangle \) to the indices \( k_1 < \ldots < k_u \) in \( LS(PB) \), a mapping \( C : \{1, \ldots, u\} \rightarrow \{k_1, \ldots, k_u\} \) is introduced. For all \( i, 1 \leq i \leq u \), \( C(i) \) is the index of bead \( eb_i \) in \( LS(PB) \). The definition of beads \( eb_1 \) to \( eb_u \) can be equivalently formulated in terms of \( LS(PB) \) as

\[ (\forall i : 1 \leq i \leq u : (\forall j : s + 1 \leq j \leq l \wedge j \notin \{k_1, \ldots, k_u\} : w_{e_j} \geq w_{e_k}) \). \]

(5.3)

The feasibility of \( LS(PB) \) implies that \( LS(SB) \ominus \langle b_{sb+1} \rangle \) is a prefix of a feasible local schedule and thus \( \neg \text{PrefixFeasibleLocalSchedule}(LS(SB) \ominus \langle b_{sb+1} \rangle) \) holds. Under the assumption that the left hand side of the implication of Theorem 5.2 holds, this implies that

\[ s(t(b_{sb+1}, LS(SB))) \geq s(t(b_{C(1)}, LS(SB))). \]

From this relation, it can be derived, using Definition 5.2, that this implies that the window start time of bead \( eb_1 \) is at most the start time of bead \( b_{sb+1} \) in \( LS(PB) \)

\[ w_{b_{C(1)}} \leq s_{b_{sb+1}}. \]

(5.4)

After bead \( b_{sb} \) and before bead \( b_{k_u} \) in \( LS(PB) \) there are \( k_u - sb - u = o \) beads not from sequence \( \langle eb_1, \ldots, eb_u \rangle \). Suppose \( r_1 < \ldots < r_o < k_u \) are the indices in \( LS(PB) \) of these beads. Local schedule \( LS(PB) \) can be transformed into a local schedule \( LS'(PB) = \langle b_1', \ldots, b_l' \rangle \)


5.2. Problem Analysis

with \( LS(SB) \oplus (eb_1, ..., eb_u) \) as a prefix. This is done by removing beads \( b_{k_1} \) to \( b_{k_u} \) from \( LS(PB) \), ordering these beads according to mapping \( C \) to obtain sequence \( (eb_1, ..., eb_u) \) and inserting this sequence after bead \( b_{k_1} \). This transformation can be represented by the following mapping between the beads of \( LS'(PB) \) and \( LS(PB) \) (see Figure 5.2 for a case where \( u = 2 \)):

\[
\begin{align*}
(\forall i: 1 \leq i \leq sb: b'_i &= b_i) \land \\
(\forall i: sb + 1 \leq i \leq sb + u: b'_i &= b_{C(i-sb)}) \land \\
(\forall i: sb + u + 1 \leq i \leq sb + u + o: b'_i &= b_{r_{i-sb-u}}) \land \\
(\forall i: sb + u + o + 1 \leq i \leq l: b'_i &= b_l).
\end{align*}
\]

It has to be proven that \( LS'(PB) \) is feasible, and thus that each bead in

![Diagram of bead sequence and mapping]

Figure 5.2: Transformation from local schedule \( LS(PB) \) into local schedule \( LS'(PB) \) in the proof of Theorem 5.2 for a case where \( u = 2 \).

\( LS'(PB) \) ends before its window end time. This is informally shown in Figure 5.2. First, beads \( b'_1 \) to \( b'_{sb} \) form partial feasible local schedule \( LS(SB) \) and thus each of these beads ends before its window end time. Second, beads \( b'_{sb+1} \) to \( b'_{sb+o} \) form the connected sequence and since \( Feasible(LS(SB) \oplus (eb_1, ..., eb_u)) \) holds, each of these beads ends before its window end time. Third, the transformation is such that for each bead \( b'_{sb+u+i}, 1 \leq i \leq o \) holds
that all the beads of sequence \(e_{b_1}, \ldots, e_{b_n}\) scheduled after \(b_r\) in \(LS(PB)\) are scheduled before bead \(b'_{s_2u+1}\) in \(LS'(PB)\). Furthermore, formula (5.4) implies that the latest time at which bead \(b'_{s_2u+1}\) is started is the start time of \(b_{s_2u+1}\). In addition, the start time of each bead \(b'_{s_2u+1}\) to \(b'_{s_2u}\) is equal to the end time of its predecessor in \(LS'(PB)\), because the bead is part of the connected sequence. Consequently, each bead \(b_{s_i}, 1 \leq i \leq o\), in \(LS(PB)\) is shifted forward by the transformation for at most the sum of the execution times of all the beads of sequence \(e_{b_1}, \ldots, e_{b_n}\) scheduled after bead \(b_r\) in \(LS(PB)\). Therefore, the end time of each bead \(b'_{s_i+u+1}, 1 \leq i \leq o\), is at most the end time of bead \(b_{s_i}\) and thus they end before \(w_e_{b_n}\). Formula (5.3) and the mapping imply that the wc of each bead \(b'_{s_i+u+1}, 1 \leq i \leq o\), is at least \(w_{e_{b_n}}\). Thus, each of these beads ends before its window end time. Finally, each bead \(b'_{s_2u+1}\) to \(b_i\) is not scheduled later as in \(LS(PB)\), and thus each of these beads ends before its window end time. This completes the proof.

\[\square\]

5.2.3 Independent Partial Local Schedules

The third theorem concerns a situation in which \(LS(SB)\) and \(UB\) are such that the window start time of each unscheduled bead is at least the end time of \(LS(SB)\). In this case, \(LS(SB)\) is called an independent partial local schedule. For convenience, the end time of a partial local schedule is defined first.

**Definition 5.3 (Local schedule end time)** Let \(SB\) be the set of scheduled beads and \(LS(SB) = \langle b_1, ..., b_{|SB|} \rangle\) a partial local schedule for \(SB\). Then, the end time of a local schedule is defined as

\[
ct(\text{LS}(SB)) = \begin{cases} 
st_{|SB|} + c_{|SB|} & \text{if } SB \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}
\]

Now, the independence property is defined.

**Property 5.5** Let \(SB\) be the set of scheduled beads and \(UB\) the set of unscheduled beads. Let \(LS(SB)\) be a partial feasible local schedule for \(SB\). Then, the independent partial local schedule property for \(LS(SB)\) and \(UB\) is defined as

\[
\text{Independent Partial Local Schedule}(LS(SB), UB) \Rightarrow 
\forall b : b \in UB : b.ws \geq ct(\text{LS}(SB)).
\]
5.2. Problem Analysis

The third theorem states that if $LS(SB)$ is independent, it is a feasibility preserving bead sequence for the empty local schedule. Property 5.3 implies that, if there is a feasible local schedule, there is a feasible local schedule with $LS(SB)$ as a prefix.

**Theorem 5.3** Let $SB$ be the set of scheduled beads and $UB$ the set of unscheduled beads. Let $LS(SB)$ be a partial feasible local schedule for $SB$. Then, the following holds:

$$\text{IndependentPartialLocalSchedule}(LS(SB), UB) \Rightarrow \text{FeasibilityPreservingBeadSequence}(\emptyset, LS(SB)).$$

**Proof** For the proof of Theorem 5.3, it is assumed that the left hand side of the implication holds. Then, it is shown that an arbitrary feasible local schedule can always be transformed into a feasible local schedule with $LS(SB)$ as a prefix. The proof presented here is informal and shows intuitively that the theorem holds.

Consider an arbitrary feasible local schedule $LS(PB) = \langle b_1, \ldots, b_l \rangle$, where $|PB| = l$. Suppose that $SB$ contains $sb$ beads and that these beads are positioned in $LS(PB)$ at indices $k_1$ to $k_{sb}$, $1 \leq k_1 < \ldots < k_{sb} \leq l$. In order to relate the indices 1 to $sb$ in $LS(SB)$ to the indices $k_1$ to $k_{sb}$ in $LS(PB)$, a mapping $C : \{1, \ldots, sb\} \rightarrow \{k_1, \ldots, k_{sb}\}$ is introduced. For all $i$, $1 \leq i \leq sb$, $C(i)$ is the index of bead $sb_i$ in $LS(PB)$. All other $l - sb$ beads in $LS(PB)$ are beads of set $UB$. Suppose, the beads of $UB$ are positioned in $LS(PB)$ at indices $r_1$ to $r_{l-sb}$, $1 \leq r_1 < \ldots < r_{l-sb} \leq l$. The independence property of $LS(SB)$ can now be equivalently formulated in terms of $LS(PB)$ as

$$\forall i : 1 \leq i \leq l - sb : \text{ws}_r_i \geq et(LS(SB)).$$

(5.5)

Local schedule $LS(PB)$ can be transformed into a local schedule $LS'(PB) = \langle b'_1, \ldots, b'_l \rangle$ with $LS(SB)$ as a prefix. This is done by removing beads $b_{k+1}$ to $b_{k_{sb}}$ from $LS(PB)$, ordering these beads according to mapping $C$ to obtain $LS(SB)$ and inserting this sequence in front of bead $b_{r_1}$. This transformation can be represented by the following mapping between the beads of $LS'(PB)$ and $LS(PB)$ (see Figure 5.3 for a case where $sb = 2$):

$$\forall i : 1 \leq i \leq sb : b'_i = b_{C(i)} \wedge$$

$$\forall i : sb + 1 \leq i \leq l : b'_i = b_{r_{i-sb}}.$$
Chapter 5. Local Scheduling

Figure 5.9: Transformation from local schedule $LS(PB)$ into local schedule $LS'(PB)$ in the proof of Theorem 5.3 for a case where $sb = 2$.

It has to be proven that $LS'(PB)$ is feasible, and thus that each bead in $LS'(PB)$ ends before its window end time. This is informally shown in Figure 5.3. First, beads $b'_l$ to $b'_m$ form partial feasible local schedule $LS(SB)$ and thus each of these beads ends before its window end time. Second, formula 5.5 implies that bead $b'_{m+1}$ starts at its window start time. Consequently, bead $b'_{m+1}$ has not been shifted forward in time by the transformation. Moreover, no bead $b'_{i+1}$, $1 \leq i \leq l - sb$, has been shifted forward in time either, because the transformation preserves the order among these beads. Hence, each bead $b'_{m+1}$ to $b'_l$ ends before its window end time. This completes the proof.

An important consequence of this theorem in conjunction with Corollary 5.1 is that each bead $b_i$ in $LS(SB)$ is a feasibility preserving bead for $LS(SB) \oplus (b_1, ..., b_{i-1})$. This is expressed by the following corollary.

**Corollary 5.2** Let $SB$ be the set of scheduled beads and $UB$ the set of unscheduled beads. Let $LS(SB)$ be a partial feasible local schedule for $SB$. Then, the following holds:

$IndependentPartialLocalSchedule(LS(SB), UB)$
5.3. A Local Scheduling Algorithm

\[(\forall i: 1 \leq i \leq |SB|:\]
\[
\text{FeasibilityPreservingBead}(\langle b_1, \ldots, b_{i-1}, b_i \rangle, b_i)
\].

5.3 A Local Scheduling Algorithm

In this section, a heuristic constructive algorithm is proposed that can solve the local scheduling problem. The algorithm takes advantage of the theorems presented in Section 5.2 and constructs a feasible local schedule by repeatedly extending a partial feasible local schedule at the end. The algorithm is depicted in Figure 5.4. The algorithm consists of an initialization part and a repetition. The algorithm maintains a partial feasible local schedule $LS(SB)$ for a set of scheduled beads $SB \subseteq PB$. The set of unscheduled beads is then defined as $UB = PB \setminus SB$. Then, the set of candidate beads $CB$ contains beads of $UB$ which are candidates to be added to $LS(SB)$. It is not guaranteed that each bead in $CB$ leads to a feasible local schedule for $PB$. On the other hand, the set of candidate beads is used to recognize beads that certainly do not lead to a feasible local schedule for $PB$.

In the initialization part of the algorithm, $SB$ is initialized to the empty set, $UB$ is initialized to $PB$ and $LS(SB)$ is initialized to the empty local schedule. Then, a set of candidate blocks $CB$ is constructed for $LS(SB)$ and $UB$. This is done according to a heuristic discussed in Section 5.3.1.

The repetition of the algorithm terminates in three cases. First, when all beads are scheduled, i.e., $SB = PB$. In this case, a feasible local schedule for $PB$ is found. Second, there are no more candidate beads, i.e., $CB = \emptyset$. In this case, if $SB \neq PB$, $LS(SB)$ cannot be extended with another bead, although there are still unscheduled beads in $UB$. Then a feasible local schedule for $PB$ is not found. Third, the number of bead scheduling attempts has reached a maximum $M_{NBSA}$. In this case, the repetition is terminated and a feasible local schedule for $PB$ is not found.

Within the repetition, a bead $sb$ is selected from $CB$, according to a heuristic discussed in Section 5.3.2. This bead is added to $LS(SB)$ and removed from $UB$. Then, if possible, the theorems and corollaries presented in Section 5.2 are applied using a procedure discussed in Section 5.3.3. Subsequently, a set of candidate beads $CB$ is constructed for the new $LS(SB)$ and the new $UB$ with the candidate set construction heuristic. At this point, it is possible that there are no candidate beads, i.e., $CB = \emptyset$, and thus $LS(SB)$
is not a prefix of a feasible local schedule for $PB$. In this case, the algorithm starts backtracking along $LS(SB) = \langle b_1, \ldots, b_n \rangle$. A backtracking bead $b_i$

**Local Scheduling Algorithm**

```plaintext
Local_Scheduling_Algorithm(in PB; out FOUND, LS(PB));
/* SB is the set of scheduled beads, UB is the set of unscheduled */
/* beads and LS(SB) is the partial feasible local schedule */
/* invariant: FeasibleLocalSchedule(LS(SB)) */
SB := ∅; UB := PB; LS(SB) := ∅;
/* invariant holds */
Construct_Candidate_Bead_Set(LS(SB), UB, CB);

nbse := 0;
/* nbse is the number of bead scheduling attempts */
while (SB ≠ PB ∧ CH ≠ ∅ ∧ nbse < MNSA) do
  nbse := nbse + 1;
  Select_Bead(LS(SB), UB, CB, sch);
  /* sch ∈ UB ∧ st(sch, LS(SB)) + sch ∈ LS(SB) */
  SB := SB ∪ \{sch\}; UB := UB \ {sch};
  LS(SB) := LS(SB) ∪ \{sch\};
  /* invariant holds */
  Construct_Candidate_Bead_Set(LS(SB), UB, CB);
  Apply_Theorems_And_Corollaries(LS(SB), UB);
  Construct_Candidate_Bead_Set(LS(SB), UB, CB);
  if (CH = ∅) then
    /* suppose LS(SB) = \langle b_1, \ldots, b_n \rangle */
    Select_Backtracking_Pair(LS(SB), UB, FOUND, a);
    if FOUND then
      SB := SB \ \{b_1, \ldots, b_n\};
      UB := UB \ \{b_1, \ldots, b_n\};
      LS(SB) := \langle b_1, \ldots, b_n \rangle;
      /* invariant holds */
      Construct_Candidate_Bead_Set(LS(SB), UB, CB);
      CB := CB \ \{b | b preferred to a
                      according to bead selection heuristic \};
    end if;
  end if;
end while;
FOUND := SB = PB;
/* FOUND ⇔ FeasibleLocalSchedule(LS(PB)) */
End.
```

**Figure 5.4**: The local scheduling algorithm.

is selected from $SB$ which probably caused the absence of candidates after $LS(SB)$. Also, an alternative bead $ab$ is selected which has to replace bead $b_i$. Selection of these beads is done with a backtracking heuristic discussed in Section 5.3.4. After selection, each bead of the sequence $\langle b_1, \ldots, b_n \rangle$ is removed from $LS(SB)$ and added to $UB$. Subsequently, a set of candidate beads $CB$ is constructed for the new $LS(SB)$ and the new $UB$ according to the candidate set construction heuristic. Note that $CB$ contains the back-
5.3. A Local Scheduling Algorithm

tracking bead $b_i$ as well as alternative bead $ab$. The selection of bead $ab$ is enforced in the next iteration of the repetition. To this end, each bead in $CB$ that is preferred to bead $ab$ according to the bead selection heuristic is removed from $CB$.

5.3.1 Candidate Set Construction

A crucial step in the algorithm presented above is the selection of the next bead to be added to $LS(SB)$. Obviously, it is desirable that a bead is selected which leads to a feasible local schedule. However, in general, this does not hold for most beads in $UB$. For instance, for a bead $b \in UB$, $LS(SB) \oplus (b)$ is not a prefix of a feasible local schedule when some other bead in $UB$ can never end before its we when it is added to $LS(SB) \oplus (b)$. To avoid the selection of such a bead, a set of candidate beads $CB \subseteq UB$ is constructed.

The construction of a candidate set if done by the procedure in Figure 5.5. A feasibility condition for $LS(SB)$, $UB$ and a bead $b \in UB$ is derived

```
Construct_Candidate_Bead_Set(in:LS(SB),UB; out:CB);

CB := \emptyset;
SLB := "list of beads $b \in UB$ sorted by non-decreasing $b$ we";

for each $b \in UB$ do
  candidate := true;
  $b_1$ := "first bead in SLB";
  sum := 0;
  while "there is a $b_2$" do
    if ($b_1 \leq b$) then
      sum := sum + $b_1$ we;
      /* sum = ($\sum b_2 \in UB \setminus \{b\} : b_2$ we $\leq b_1$ we $= b_2$ we \(\leq \)) */
      candidate := candidate $\wedge$ (sum $\leq b_1$ we $= \{w(b,LS(SB)) + b$ we$\}$);
    end if:
    $b_2$ := "next bead after $b_1$ in SLB";
  end while;
  /* candidate $\Rightarrow FeasibleLocalSchedule(LS(SB) \oplus (b))$ */
  if (candidate) then
    $CB := CB \cup \{b\}$;
  end if:
end for;
/* $\forall b \in CB$ : FeasibleLocalSchedule(1S(SB) \oplus (b)) */
End.
```

Figure 5.5: The procedure that constructs a set of candidate beads $CB$ for a partial feasible local schedule $LS(SB)$ and a set of unscheduled blocks $UB$.

which is satisfied if $LS(SB) \oplus (b)$ is a prefix of a feasible local schedule. Then, a bead $b \in UB$ is a candidate bead if the feasibility condition for
$LS(SB)$, $UB$ and $b$ is satisfied. Thus, set $CB$ can be formulated as

$$CB = \{ b \in UB \mid FeasibilityCondition(LS(SB), UB, b) \}.$$  

Consequently, for each bead $b \in UB \setminus CB$, the feasibility condition is not satisfied, and thus $LS(SB) \oplus (b)$ is not a prefix of a feasible local schedule. On the other hand, for each head $b \in CB$, the feasibility condition is satisfied, and thus $LS(SB) \oplus (b)$ may be a prefix of a feasible local schedule.

The feasibility condition requires that when $b \in UB$ is added to $LS(SB)$, the time interval between the end of the execution of $b$ and the window end of each bead $b_1$ in $UB$ is large enough to accommodate all beads $b_2$ in $UB \setminus \{b\}$ with $b_2,we \leq b_1,we$

$$FeasibilityCondition(LS(SB), UB, b)$$

$$\Rightarrow$$

$$(\forall b_1 \in UB :$

$$(\sum b_2 \in UB \setminus \{b\} : b_2,we \leq b_1,we : b_2,e)$

$$\leq b_1,we - (s_t(b, LS(SB)) + b,e)$$

$$.\)

Note that each candidate bead ends before its window end time when it is added to $LS(SB)$. Thus, the partial local schedule remains feasible.

The candidate set construction procedure first sorts all beads of $UB$ by non-decreasing window end time. This is done by a standard sorting procedure that has a time-complexity less than $O(|PB|^2)$. Then, for each bead in $UB$, the procedure passes forwards through the sorted list of beads only once, while maintaining and checking the execution sum. A bead is added to $SB$ if it passes all its checks. Therefore, the entire procedure has a time-complexity $O(|PB|^2)$.

### 5.3.2 Bead Selection

In the repetition of the algorithm, a candidate bead is selected first. This is done using the procedure of Figure 5.6. This procedure uses a heuristic that selects a bead with the smallest start time after $LS(SB)$ among all candidates. Thereby, the amount of idle time introduced after $LS(SB)$ is as small as possible. If there are more candidates with the smallest start time after $LS(SB)$, the heuristic selects the one with the smallest window end
5.3. A Local Scheduling Algorithm

```
Select_Bead((in:LS(SB), CB; out:scb))
bestat := oo;
bestue := oo;
for each b ∈ CB do
    if (st(b, LS(SB)) < bestat) ∨
        (st(b, LS(SB)) = bestat ∧ b.w ≤ bestue) then
        bestat := st(b, LS(SB));
bestue := b.w;
        scb := b;
    endif;
end for;
/* scb ∈ CB */
End.
```

Figure 5.6: The procedure that selects a bead scb from the set of candidate beads CB based on partial feasible local schedule LS(SB).

time. Summarizing, the heuristic selects the bead scb such that

\[
\begin{align*}
(\forall b ∈ CB : & \quad st(b, LS(SB)) > st(scb, LS(SB)) \lor \quad st(b, LS(SB)) = st(scb, LS(SB)) ∧ b.w ≤ scb.wc \\
& ).
\end{align*}
\]

(5.6)

Note that when the window start time of each bead in CB is larger than the end time of LS(SB), an idle time interval, i.e., a gap in the local schedule results. A gap is a time interval in a local schedule in which no bead is scheduled. The bead selection procedure selects a candidate by checking each bead in CB. Therefore, the procedure has a time-complexity \(O(|PB|)\).

As described in the previous section, for each bead \(b ∈ UB \setminus CB\), \(LS(SB) ⊕ (b)\) is not a prefix of a feasible local schedule. Then, since scb is a bead with the smallest start time after LS(SB)

\[
(\forall b ∈ UB \setminus \{scb\} : st(b, LS(SB)) < st(scb, LS(SB)) ; \quad \neg Prefix\ Feasible\ Local\ Schedule(\ LS(SB) ⊕ (b)) \).
\]

(5.7)

This formula is equivalent to the left hand side of the implication in Theorem 5.1 where scb is substituted for cb. Also, this formula is equivalent to the left hand side of the implication in Theorem 5.2 where scb is substituted for cb1. Thus, the left hand side of the implications in both theorems are always
satisfied when beads are selected by the heuristic presented above. In the next section, it is described how the theorems and corollaries presented in Section 5.2 are applied when bead \( sb \) is added to \( LS(SB) \).

### 5.3.3 Application of Theorems and Corollaries

When the selected bead \( sb \) is added to \( LS(SB) \) and sets \( SB \) and \( UB \) are adjusted accordingly, the algorithm applies the three theorems and two corollaries of Section 5.2 if possible. This is done using the procedure of 5.7. This

```plaintext
Apply_Theorems_And_Corollaries(init:LS(SB),UB):
/* suppose LS(SB) = (b_1,...,b_n) */
/* apply Theorem 5.1 */
smw := ∅;
for each \( b \in UB \) do
    smw := Min(smw,b.we);
/* smw = (Min b ∈ UB : b.we) */

for each \( n r w := \{ smw, b \in UB : b \} \) do
/* /n wr := \{ smw, b ∈ UB : b \} */

b_n,frb := FeasibilityPreservingBead((b_1,...,b_{n-1}),b_n);
/* apply Theorem 5.2 */

if b_n,frb then
    t := n;
while (i > 1) ∧ (b_{i-1}.st + b_{i-1}.c = b_i.st) ∧ (b_{i-1}.we ≤ smw) do i := i - 1;
/* ConnectSequence((b_1,...,b_{i-1}),(b_i,...,b_n)) */

/* (∀ j : i ≤ j ≤ n : (∀ k ∈ UB : k.we ≥ b_j.we)), then Theorem 5.2 implies */
/* FeasibilityPreservingBeadSequence((b_1,...,b_{i-1}),(b_i,...,b_n)) */
/* apply Corollary 5.1 */

for each \( j : i ≤ j ≤ n \) do b_i,frb := true;
/* (∀ j : i ≤ j ≤ n : FeasibilityPreservingBead((b_1,...,b_{i-1}),b_j) */
end if;
/* apply Theorem 5.3 */

if ind := true:
for each \( b \in UB \) do
    ind := ind ∧ (ct(LS(SB)) ≤ b.we);
/* ind ∈ IndependentPartialLocalSchedule(LS(SB),UB) */

/* thus Theorem 5.3 implies */
/* ind ∈ FeasibilityPreservingBeadSequence(),LS(SB) */

if b_n,frb then
/* apply Corollary 5.2 */

for each \( j : 1 ≤ j ≤ n \) do b_i,frb := true;
/* (∀ j : 1 ≤ j ≤ n : FeasibilityPreservingBead((b_1,...,b_{i-1}),b_j) */
end if;
/* (∀ i : 1 ≤ i ≤ n : b_i,frb ⇒ FeasibilityPreservingBead((b_1,...,b_{i-1}),b_i) */
end.
```

**Figure 5.7:** The procedure that applies the three theorems of Section 5.2 based on the current \( LS(SB) \) and \( UB \).

The procedure first tries to apply Theorem 5.1 to the last bead of \( LS(SB) \). As described in the previous section, the left hand side of this theorem is always
satisfied. Thus, if $b_n$ has a window end time at most the window end times of all beads in $UB$, the theorem implies that $b_n$ is a feasibility preserving bead.

Second, the procedure tries to apply Theorem 5.2 to the end of $LS(SB)$. Therefore, suppose that $LS(SB) = (b_1, ..., b_n)$ ends with a connected sequence $(b_i, ..., b_n)$. Thus, $ConnectedSequence((b_1, ..., b_{i-1}), (b_i, ..., b_n))$ holds. Also, suppose that beads $b_i$ to $b_n$ have window end times at most the window end times of all beads in $UB$. Then, the assumptions of Theorem 5.2 hold. As described in the previous section, the left hand side of this theorem is always satisfied. Thus, the theorem implies that sequence $(b_i, ..., b_n)$ is a feasibility preserving bead sequence after $(b_1, ..., b_{i-1})$. In addition, Corollary 5.1 implies that beads $b_i$ to $b_n$ are feasibility preserving.

Finally, the procedure tries to apply Theorem 5.3 to $LS(SB)$. Therefore, the procedure checks whether the end time of $LS(SB)$ is at most the window start time of each bead in $UB$. If so, Theorem 5.3 implies that $LS(SB)$ is a feasibility preserving bead sequence. In addition, Corollary 5.2 implies that all beads in $LS(SB)$ are feasibility preserving.

The procedure assigns a value to an attribute $b_i.fpb$ for each bead $b_i$ in $LS(SB)$, such that $b_i.fpb \Rightarrow FeasibilityPreservingBead((b_1, ..., b_{i-1}), b_i)$ holds. The procedure applies the theorems and corollaries by checking the beads in $UB$ and $LS(SB)$ several times. Therefore, the procedure has a time-complexity $O(|PB|)$.

### 5.3.4 Backtracking

When the algorithm has added a bead to $LS(SB)$, a set of candidate beads is constructed for the new $LS(SB)$ and the new $UB$. When there are no candidates, $LS(SB)$ is not a prefix of a feasible local schedule. At this point, infeasibility has occurred at the end time of $LS(SB)$ and the algorithm starts backtracking along $LS(SB)$ using the procedure of Figure 5.8. This procedure selects a backtracking bead $b_i$ in $LS(SB) = (b_1, ..., b_n)$, that is likely to have caused the absence of candidates. An alternative bead $ab$ that has to replace $b_i$ is also selected. After the procedure has selected these two beads, the algorithm removes all beads $b_i$ to $b_n$ from $LS(SB)$, adds them to $UB$, and ensures that bead $ab$ is subsequently added to $LS(SB)$ in the next iteration of the repetition.

The main idea of the backtracking procedure is to try to select a backtracking bead $b_i$ and an alternative bead $ab$ such that the algorithm “makes progress”. This means that by backtracking with $b_i$ and $ab$ infeasibility is
Select Backtracking_Pair(in:LS(SB),UB; out:FOUND,b,ab);
  /* try to find a progress making pair of beads for LS(SB) */
  Find_Progress_Making_Beads(LS(SB),UB,FOUND,b,ab);
  /* FOUND or b, and ab are found */
  /* suppose LS(SB) = (b_1, ..., b_n) */
  j := n - 1;
  while ~FOUND ∧ (j ≥ 1) do
    if (b_j, st + b_j, st < b_{j+1}, st)
      then Find_Progress_Making_Beads ((b_1, ..., b_j), UB ∪ {b_{j+1}, ..., b_n}, FOUND,b,ab);
    j := j - 1;
  end while;
  if ~FOUND then Find_Latest_Backtracking_Pair(LS(SB),UB,FOUND,b,ab);
  /* FOUND or b is the backtracking bead and ab the alternative bead */
End.

Figure 5.8: The procedure that selects a backtracking bead b_i and an alternative bead ab for partial feasible local schedule LS(SB).

not encountered before or at the end time of the current LS(SB) again.
Therefore, the procedure first tries to find a b_i and an ab that make progress for LS(SB). This is done by a procedure discussed in Section 5.3.4.1. By backtracking with these two beads the infeasibility is hopefully solved. If there is not such a pair for LS(SB), the procedure makes use of the gaps in LS(SB), because a gap might indirectly cause the infeasibility. The backtracking procedure passes backwards through LS(SB) and checks gaps. By passing backwards through LS(SB), the search tree is pruned as little as possible. Per gap, the backtracking procedure tries to find a b_i and an ab that make progress for the part of LS(SB) before the gap. This is also done by the procedure discussed in Section 5.3.4.1. By backtracking with these two beads the gap is hopefully closed and the infeasibility is hopefully solved. Finally, if there is not such a pair for each gap in LS(SB), the backtracking procedure tries to find the latest bead b_i in LS(SB) that is not feasibility preserving. This is done by a procedure discussed in Section 5.3.4.2. In addition, this procedure tries to find an alternative bead after (b_{i+1}, ..., b_{n-1}) according to the bead selection heuristic expressed by formula 5.6. Thereby, the search tree is pruned as little as possible.

5.3.4.1 Progress Making Beads

Finding a backtracking bead b_i and an alternative bead ab that make progress for a partial feasible local schedule LS(SB) = (b_1, ..., b_n) with a set of un-
scheduled beads $UB$ is done by the procedure in Figure 5.9. First of all, it
can be easily verified that bead $b_i$ does not cause infeasibility if it is feas-
bility preserving. Thus, such a bead must not be selected as backtracking
bead. As introduced in Section 5.3.3, if attribute $b_i.fpb$ holds, $b_i$ is feasibility
preserving. Second, in order to terminate, the alternative bead $ab$ must
be a candidate bead after partial local schedule $\langle b_1, ..., b_{i-1} \rangle$ that is not yet
selected to be added to $\langle b_1, ..., b_{i-1} \rangle$. For convenience this is expressed by a
predicate $NotYetSelected(\langle b_1, ..., b_{i-1} \rangle, ab)$. According to the bead selection
heuristic expressed by formula (5.6), beads with smaller start times are se-
lected first. Therefore, the start time of $ab$ after $\langle b_1, ..., b_{i-1} \rangle$ is at least the
start time of $b_i$.

The heuristic used in the procedure is based on the following discussion.
The possibility that progress is made with $b_i$ and $ab$ can be determined by
analyzing the expected behaviour of the algorithm after backtracking on
these beads. The behaviour of the algorithm is analyzed based on a number
of observations. The first observation is that when bead $ab$ is added to
$\langle b_1, ..., b_{i-1} \rangle$, it is very likely that the algorithm then selects the original
candidate beads $b_i, b_{i+1},$ and so on. Suppose that $ab$ is one of the beads $b_{i+1}$
to $b_n$. Then, the algorithm will construct (1) a partial local schedule $LS(SB)$
in which bead $ab$ is scheduled earlier and (2) the initial set of unscheduled
beads $UB$. Since the start time of $ab$ is at least the start time of $b_i$, as
discussed above, the end time of this partial local schedule is at least the
end time $et(LS(SB))$ of the initial $LS(SB)$, and thus the same infeasibility
is encountered again. Therefore, $ab$ should be a bead of the initial $UB$.

A second observation is that it is desirable to analyze a partial local
schedule with an end time larger than the end time $et_{LS(SB)}$ of the initial
$LS(SB)$. Thereby, the expected behaviour is analyzed after the algorithm
has made progress. On the other hand, it is desirable to analyze a situation
in which not too much progress is made. Thereby, the expected behaviour is
analyzed close to the point in time where the infeasibility occurred. There-
fore, the partial local schedule $\langle b_1, ..., b_{i-1}, ab, b_i, ..., b_k \rangle$ is analyzed, where
$k \leq n$ is the largest index such that the start time of $b_k$ is at most the end
time $et(LS(SB))$ of $LS(SB)$.

A third observation is that the start time of $ab$ after $\langle b_1, ..., b_{i-1} \rangle$, should
equal the start time of $b_i$ in $LS(SB)$. Otherwise, an idle interval results
between the end time of $b_{i-1}$ and the start time of $ab$ and it is likely that
infeasibility occurs again. A fourth observation is that the execution time
of $ab$ should be larger than the execution time of $b_i$. This way, the end
time of $\langle b_1, ..., b_{i-1}, ab \rangle$ is larger than the end time of $\langle b_1, ..., b_i \rangle$ and thus
Find_Progress_Making_Beads(in:LS(SB),UB; out:FOUND,b1,ab):
/* suppose LS(SB) = (b1,...,bn) */
found := false;
j := n;
while ¬found ∧ (j ≥ 1) do
  if ¬bj,jk then
    Construct_Candidate_Bead_Set ((b1,...,bj−1),UB ∪ {bj,...,bn},CB);
    SLB := "list of beads CB sorted according to the bead selection heuristic";
    k := "first bead in SLB after bj";
    while ¬found ∧ "there is a b" do
      if (b ∈ UB) ∧ (str(b,(b1,...,bj−1)) = bj,at) ∧ (b.r > bj.r) then
        LS := (b1,...,bj−1,bj);
        k := j;
        while (str(bk,LS) ≤ str(LS(SB))) ∧ (k ≤ n) do
          LS := LS ∪ bk;
          k := k + 1;
        end while;
        /* LS is the expected partial local schedule */
        Construct_Candidate_Bead_Set (LS,UB ∪ {bk,...,bn},CB);
        found := CB ∉ δ;
        if found then
          b1 := bj;
          ab := b;
        end if;
      end if;
    k := "first bead after b in SLB";
  end while;
end if;
j := j − 1;
end while;
/* found ⇒ Progress_Making_Beads(LS(SH),UB,b1,ab) */
End.

Figure 5.9: The procedure that selects a backtracking bead bj and an alternative bead ab that make progress for a partial feasible local schedule LS(SB) and a set of unscheduled beads UB.

more progress is made. Finally, a fifth observation is that for a feasible local schedule there has to be a candidate bead after (b1,...,bj−1,ab,bj,...,bn).

Based on the discussion above, the progress making beads property can be defined for a backtracking bead bj and an alternative bead ab. For convenience, CB(LS(SB),UB) is the set of candidate beads after partial local schedule LS(SB) given a set of unscheduled beads UB.

**Property 5.6** Let SB be the set of scheduled beads and UB the set of unscheduled beads. Let LS(SB) = (b1,...,bn) be a partial feasible local schedule for SB. Let ab be a bead that is a candidate after (b1,...,bj−1). Let (b1,...,bj−1,ab,bj,...,bk) a partial local schedule where k ≤ n is the largest
index such that the start time of \( b_k \) is at most \( et(LS(SB)) \). Then, the progress making beads property for \( LS(SB), UB, b_i \) and \( ab \) is defined as

\[
ProgressMakingBeads(LS(SB), UB, b_i, ab) \\
\Leftrightarrow \\
\neg b_i.fpb \land \\
NotYetSelected((b_1, ..., b_{i-1}, ab) \land(ab \in UB \land st(ab, (b_1, ..., b_{i-1})) = b_i.st \land ab.e > b_i.e \land \\
CB((b_1, ..., b_{i-1}, ab, b_i, ..., b_k), (UB \cup \{b_{k+1}, ..., b_n\}) \setminus \{ab\}) \neq \emptyset).
\]

The procedure searches for a progress making pair of beads by passing backwards through \( LS(SB) \). For each proper \( b_i \), the procedure checks alternative beads \( ab \) in order of decreasing preference according to the bead selection heuristic. The procedure terminates as soon as a progress making pair of beads has been found or when each pair of beads does not satisfy Property 5.6. This way, the search tree is pruned as less as possible.

In the worst case, the candidate set construction procedure is called for each pair of beads \( b_i \) and \( ab \) that is checked. Therefore, the time-complexity of the procedure for finding progress making beads is \( O(|PB|^4) \).

### 5.3.4.2 Latest Backtracking Pair

Finding the latest pair of a backtracking bead \( b_i \) and an alternative bead \( ab \) for a partial feasible local schedule \( LS(SB) = (b_1, ..., b_n) \) with a set of unscheduled beads \( UB \) is done by the procedure in Figure 5.10. This procedure selects as backtracking bead the latest bead \( b_i \) in \( LS(SB) \) that is not feasibility preserving. In addition, the procedure selects as alternative bead the next unselected bead after \( (b_1, ..., b_{i-1}) \) according to the bead selection heuristic. Thereby, the search tree is pruned as little as possible. The next preferred bead property is defined as follows.

**Property 5.7** Let \( SB \) be the set of scheduled beads and \( UB \) the set of unscheduled beads. Let \( LS(SB) \) be a partial feasible local schedule for \( SB \). Let \( b \) be a candidate bead in \( CB(LS(SB), UB) \). Let \( ab \) also be a candidate bead in \( CB(LS(SB), UB) \). Then, the next preferred bead property for \( LS(SB), b \) and \( ab \) is defined as

\[
NextPreferredBead(LS(SB), b, ab)
\]
Find_Latest_Backtracking_Pair(in: \( LS(SB), UB \); out: FOUND, \( b, ab \)):

/* suppose \( LS(SB) = (b_1, ..., b_n) \) */
FOUND := false;
\( j := n \);
while not FOUND \( \land (j \geq 1) \) do
  if not \( j \), \( f \neq b \) then
    Construct_Candidate_Bead_Set \((b_1, ..., b_{j-1}), UB \cup \{b_j, ..., b_n\}, CB)\);
    SLB := "list of beads CB sorted according to the bead selection heuristic";
    FOUND := "\( b_j \) is not the last bead in SLB";
    if FOUND then
      \( b_i := b_j \);
      \( ab := \) "first bead after \( b_j \) in SLB";
      end if;
  end if;
\( j := j - 1 \);
end while;
/* FOUND := \( b, f \neq b \land Next_Preferred_Bead(LS(SB), b, ab) */
End.

Figure 5.10: The procedure that selects the latest backtracking bead \( b_i \) and alternative bead \( ab \) for a partial feasible local schedule \( LS(SB) \) and a set of unscheduled beads \( UB \).

\[\Rightarrow\]
NotYetSelected(LS(SB), ab) \land 
\((\forall b' : b' \in CB \setminus (LS(SB), UB) \land \text{NotYetSelected}(LS(SB), b') \land \)
\(st(b', LS(SB)) \geq st(b, LS(SB)) \land 
st(b', LS(SB)) > st(ab, LS(SB)) \lor 
(st(b', LS(SB)) = st(ab, LS(SB)) \land b' \text{we} \geq ab \text{we})\).

In the worst case, the candidate set construction procedure is called for each bead \( b_i \) in \( LS(SB) \). Therefore, the time-complexity of the procedure for finding the latest pair of backtracking beads is \( O(|PB|^3) \).

Based on the time-complexities of the progress making beads procedure and the latest backtracking pair procedure, it can be concluded that the entire backtracking procedure has a time-complexity \( O(|PB|^6) \). Then, the local scheduling algorithm has a time-complexity \( O(|PB|^6) \). This is a worst case upper bound on the computation time of the algorithm. Performance measurements presented in Chapter 6 show that the average computation time of the algorithm is quadratic in the number of beads.
5.4 Conclusion

In this chapter, a heuristic constructive algorithm is presented that can solve the local scheduling problem of the third step of the scheduling approach. The algorithm uses an enhanced version of the earliest deadline first strategy that avoids infeasibility by using a strong feasibility condition. A backtracking heuristic for resolving infeasibility is formed by three theorems that identify scheduling decisions that did not cause the infeasibility. Results of performance measurements are presented in Chapter 6. They show that the algorithm is quite efficient with respect to the percentage of problem instances for which a solution can be found. In addition, the computation time on a standard PC needed to find a solution is acceptable even for large instances.
Chapter 6

Performance Measurements

In this chapter, performance measurements of the pre-run-time scheduling approach are presented and discussed. The algorithms and the entire approach were tested using input from a hardware architecture generator as well as an application generator. Each generator has a number of input parameters that determine the most common characteristics of the generated hardware architectures and applications. The performance for several combinations of the input variables of the two generators are briefly discussed in this chapter. The results only give a rough indication of the performance of the entire pre-run-time scheduling approach. The local scheduling problem is a well-known problem on its own. Therefore, in order to show the usefulness of the local scheduling algorithm in isolation, it was tested with an exhaustive set of problem instances. These instances were generated by a separate bead set generator that can vary all characteristics of the local scheduling problem.

6.1 Input Generators

Two input generators were implemented, in order to test the algorithms and the entire pre-run-time scheduling approach. One of these generates hardware architectures, the other one generates applications according to the model described in Chapter 2.
6.1.1 Hardware Architecture Generator

In general, there are numerous possible parameters of a distributed hardware architecture. In the model used throughout this thesis this number is large as well. Since it is not practical to perform tests for each possible combination only the size of the hardware architecture is varied.

First of all, the generator can only generate architectures that consist of a set of multiprocessors. The number of processors in a multiprocessor is between 1 and 5, while the total number of processors is set by a parameter \( NPR \) (\( \geq 1 \)). This parameter determines the size of the hardware architecture. All processors in a multiprocessor are connected by a single asynchronous bus with a bandwidth of 25M bytes per second. Processors in different multiprocessors can be connected directly by a point-to-point bidirectional synchronous link. Each link has a bandwidth of 2.5M bytes per second. The link topology is such that there is a link between each pair of processors in different multiprocessors.

The maximum number of devices that can be connected to a processor is set to 4. Consequently, the maximum number of devices in the architecture equals \( 4 \times NPR \). However, the architecture only contains 50% of this maximum. It consists of 45% sensors, 45% actuators and 10% disks. The unit size of a piece of data involved in a blocking device access is 1 byte. The constant blocking time of a sensor equals 650 microseconds (\( \mu s \)). Since a blocking device access to a sensor does not depend on the amount of data involved, the variable blocking time of a sensor equals 0. A device access to an actuator is non-blocking, because during the access data is read from or written to the device registers in a few processor cycles. Therefore, the constant and variable blocking time of an actuator are 0. Finally, the constant blocking time of a disk is 25 milliseconds (ms), while the variable blocking time of a disk 1.6\( \mu s \).

An architecture is generated by first clustering the \( NPR \) processors randomly into multiprocessors. Second, links are assigned such that there is a link between each pair of processors in different multiprocessors. Third, a route between each pair of multiprocessors is determined. The route from a multiprocessor \( m_i \) to a multiprocessor \( m_j \) consists of a link from \( m_i \) to an intermediate multiprocessor \( m_k \) followed by the route from \( m_k \) to \( m_j \). Thereby, \( m_k \) is the multiprocessor that is closest to \( m_j \) among all multiprocessors that have a link with \( m_j \). Whenever there is more than one link between two multiprocessors, one of them is chosen randomly to be contained in the route. Fourth, routes between each pair of processors are determined.
The route between two processors in the same multiprocessor consists of the bus of the multiprocessor. The route from a processor in \( m_i \) to a processor in \( m_j \) consists of the route of links and processors from \( m_i \) to \( m_j \). When the route contains two successive processors in the same multiprocessor, the multiprocessor bus is added to the route. Finally, the devices are generated according to the restrictions described and evenly distributed over all processors.

### 6.1.2 Application Generator

The application generator generates the static and dynamic structure of an application for a particular hardware architecture. First, a set of periodic activities that make method calls and blocking device accesses is generated. Then, the processes and the graph of beads with constraints is derived from the activities. An application is generated such that there is a correct process assignment and a feasible bead schedule for the particular hardware architecture.

Since the number of variables of an application is quite large, it is not practical to perform tests for each possible combination of these variables. In the application generator only four quantities can be varied: (1) the size of the processing beads, (2) the ratio of method calls and blocking device accesses, (3) the tightness of the absolute timing constraints of the beads, and (4) the number and tightness of relative timing constraints.

First, a set of periodic activities is generated, each consisting of a sequence of beads. This is done by repeatedly generating a periodic activity for each processor as described in the next paragraph. Thus, in each period the activity starts and ends at that processor. Thereby, the generator maintains for each processor a schedule that contains the beads already generated for that processor. If there is not enough idle time in the schedule of the processor to generate an activity that consists of a single bead, the processor is called “full”. The generator continues generating activities until all processors are full. The execution time of a bead is at least a parameter \( MINE \) and at most 1 second (s), while the time resolution is 1ms. The least common multiple of all periods LCM is set to 10s.

A periodic activity is generated for a processor by first selecting a period that is a divider of LCM and at least the average execution time of a processing bead \((MINE+1s)/2\). This is done to enable the execution of several beads within the period. When a period is selected, the generator randomly decides whether an optional deadline has to be generated. If so,
a random deadline that is at most the period is selected. The deadline is
at least the average bead execution time or at least half of the period, to
enable the execution of several beads before the deadline. Then, an activity
that consists of a single bead with the selected period and deadline is
generated on the processor. This might not be possible because there is not
enough idle time on the processor. In this case, the processor is full and
a periodic activity can not be generated. Otherwise, the structure of the
activity is refined by inserting either (1) a method execution or (2) a block-
ing device access. Thereby, additional preemption points are inserted in the
activity which results in more beads. A parameter $PC$ determines the ratio
of method calls and blocking device accesses. As described below, a called
method can be assigned to another processor and thus the activity switches
to that processor. The size of parameter data for a call or return is a random
value between 0 and 5 bytes. The beads of the refined activity are generated
as described in the next paragraph. It might not be possible to insert these
beads in the schedules of the processors involved such that the deadline is
met. In this case, the refined activity can not be generated. Refinement con-
tinues until the beads of the activity can not be generated anymore. Finally,
consistency constraints for blocking device accesses and communications are
added between beads in the generated activity and beads in all other already
generated activities.

The sequence of beads and precedence relations that define the structure
of an activity is generated as follows. First, a device is randomly assigned
to each method in which a blocking device access is performed. Hence, all the
beads of the method are implicitly assigned to the processor to which the
device is assigned. The size of data involved in a blocking device access to
a sensor and an actuator is usually small and is therefore at most 5 bytes.
Usually, only small amounts of data are read from or written to a disk.
Therefore, the size of data involved in a blocking device access to a disk
is a random value between 1 and 100 bytes. Then, each method without
a blocking device access is assigned to a processor. Thereby, it is tried
to evenly distribute the beads over the processors. When all methods are
assigned to processors, communication beads via the intermediate routes
are inserted to represent parameter passing. These communication beads
are only inserted to ensure that an application is generated for which a
feasible process assignment and schedule exists. For testing of the process
assignment algorithm and the entire scheduling approach, these beads are
removed from the application. The execution times of the communication
beads are calculated using the parameter data size and the bandwidths of
6.1. Input Generators

the communication media. The earliest start time of a bead is set to the start time of the instance of the periodic activity to which the bead belongs. The deadline of the bead is set to the start time of the instance plus the deadline of the activity. Then, a start time is generated for each bead of the first instance of the periodic activity. This is done by passing in a forward direction through the sequence of beads of the activity. Per processing bead, the generator searches in the schedule of the processor involved for an interval with a length that is at least $MINE$. This is done in a way that the bead can be scheduled such that precedence relations are satisfied, device consistency is guaranteed and the deadline is met. If such an interval is found, the start time of the bead is set to the lower bound of this interval. The execution time of the bead is set to the minimum of the length of the interval and a random value between $MINE$ and 1s. Per communication bead, the generator searches in the schedule of the processor involved for an interval with a length that is at least the execution time. This is done in a way that the bead can be scheduled such that precedence relations are satisfied, consistency of the medium involved is guaranteed and the deadline is met. If such an interval is found, the start time of the bead is set to the lower bound of the interval. Finally, a start time is generated in a similar way for each bead in all other instances of the periodic activity.

When a set of periodic activities is generated, the methods are grouped into processes. This is done such that all methods that access the same device are grouped into a single object. Each method that does not access a device forms an object in itself. Then, the objects are randomly clustered into processes. In order to vary the processor utilization and the tightness of the absolute timing constraints of the beads, the generator can adjust the execution times. This is done by reducing the execution time of each processing bead to a percentage $PET$ of the bead’s initial execution time. The parameter $PET$ influences the tightness of the absolute timing constraints of the beads.

Finally, relative timing constraints are generated between beads of different activities. This is done such that the number of additional relative timing constraints is as close as possible to a percentage parameter $PAR$ of the number of processing beads in the application. A relative timing constraint between two beads with different start times is directed from the bead with the smallest start time to the other one. A relative timing constraint between two beads with identical start times is directed such that there are no cycles in the execution DAG. The offset distance between the smallest and largest offset of the relative timing constraint is a random value between
0 and 100ms. The smallest offset is then set to the difference between the start times of the two beads minus half of the offset distance. If the smallest offset is negative, it is set to 0. The largest offset is set to the smallest offset plus the selected offset distance.

### 6.2 Process Assignment Performance

The performance measures of the process assignment algorithm for a set of applications are (1) the percentage $PPF$ of applications for which a correct process assignment is found, and (2) the average amount of backtracking $APB$ that was necessary to find these process assignments. The amount of backtracking for a single application is expressed as the number of times reassignment is necessary divided by the number of processes.

The process assignment algorithm was tested for three different hardware architectures: the first with $NP_R = 5$, the second with $NP_R = 10$, and the third with $NP_R = 15$. Applications for an architecture were generated for the following parameter values of the application generator: $MINE \in \{100\text{ms}, 250\text{ms}, 500\text{ms}\}$, $PC \in \{50\%, 75\%, 100\\}$, $PET \in \{10\%, 50\%, 100\\}$, and $PAR \in \{0\%, 50\%, 100\%\}$. This resulted in applications with up to 75 processes and up to 400 beads. Since sequences of beads are generated, each bead has at most one successor. Therefore, the number of precedence relations was at most 400. Since the number of additional relative timing constraint is at most the number of beads, the total number of relative timing constraints was at most 800. Since a substantial number of beads are communication beads, the total number of consistency constraints was at most 2500. Per generated architecture, 25 applications were generated for each combination of the four parameters. Finally, for each set of 25 applications, the two performance measures of the process assignment algorithm were determined. Thereby, the parameters of the process assignment algorithm, as introduced in Chapter 3, were set as follows. The maximum number of attempts $MNA$ to assign a particular process was set to the number of processors $NP_R$. The maximum number of candidate processors $NCPR$ was set to 2 in order to restrict the computation time. In order to evenly weight the communication and parallelism objectives, $\omega_c$ and $\omega_\rho$ were both set to 0.5. The results of the measurements per hardware architecture are depicted in Table 6.1 to Table 6.3.

The most important observation is that $PPF = 100\%$ and $APB = 0$ for almost all tested combinations. This is not surprising, because the number of
6.2. Process Assignment Performance

devices is an average of the maximum and thus a correct process assignment
is easy to find. Only when parameter \( P_E \) is large, sometimes \( P_F \) is

<table>
<thead>
<tr>
<th>MIN : PC</th>
<th>P_E</th>
<th>P_A</th>
<th>P_A_P</th>
<th>P_A</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>50</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>500</td>
<td>75</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.1: Results of performance measurements of the process assignment
algorithm for a hardware architecture with \( N_P R = 5 \).

<table>
<thead>
<tr>
<th>MIN : PC</th>
<th>P_E</th>
<th>P_A</th>
<th>P_A_P</th>
<th>P_A</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
<th>P_A_P</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>50</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 6.2: Results of performance measurements of the process assignment
algorithm for a hardware architecture with \( N_P R = 10 \).

less than but close to 100\% and \( A_P \) is larger than but very close to 0.
Thus, in such a case, the algorithm can not find a process assignment for
a few applications. This probably occurs when too much communication is
inserted in an activity and the absolute timing constraints on the beads of the activity can not be satisfied in isolation. This is likely to happen when parameter \( P_{TE} \) is large because then the absolute timing constraints on the beads are very strict.

<table>
<thead>
<tr>
<th>MINR</th>
<th>PC</th>
<th>( P_{TE} )</th>
<th>( P_{AR} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>50</td>
<td>100 100</td>
<td>100 100</td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>0    0</td>
<td>0 0</td>
</tr>
<tr>
<td>250</td>
<td>35</td>
<td>100 100</td>
<td>100 100</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>0    0</td>
<td>0 0</td>
</tr>
<tr>
<td>150</td>
<td>20</td>
<td>100 100</td>
<td>100 100</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>0    0</td>
<td>0 0</td>
</tr>
<tr>
<td>75</td>
<td>15</td>
<td>100 100</td>
<td>100 100</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>0    0</td>
<td>0 0</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>100 100</td>
<td>100 100</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0    0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Table 6.3: Results of performance measurements of the process assignment algorithm for a hardware architecture with \( N_{PR} = 15 \).

Another observation is that the performance of the process assignment algorithm seems to be independent of the number of processors. This is not surprising, because the density of the interconnection topology does not change. The density of the interconnection topology determines the average length of the routes and thus the amount of communication to be inserted. Thus, it has influence on whether the absolute timing constraints on the beads of an activity can be satisfied in isolation.

### 6.3 Window Assignment Performance

Two performance measures were defined for the window assignment algorithm for a set of applications assigned to a particular hardware architecture. These are (1) the percentage \( PW_{F} \) of applications for which a presumably feasible window assignment is found, and (2) the average amount of backtracking \( AW_{R} \) that was necessary to find these window assignments. The amount of backtracking for a single application is expressed as the number of times the windows have to be adjusted in the Satisfy_Feasibility_Constraint procedure, as given in Figure 4.8, divided by the number of beads.
6.3. Window Assignment Performance

The window assignment algorithm was tested for the same three hardware architectures as the process assignment algorithm. Also the same combinations of parameters $MINE$, $PC$, $PET$ and $PAR$ were used. Per generated architecture, 25 applications were generated for each combination of these four parameters. For each set of 25 applications, the two performance measures of the window assignment algorithm were determined. The parameters of the window assignment algorithm, as introduced in Chapter 3, were set as follows. The maximum number of iterations of the adjustment loop $MINI$ (see Figure 4.7) was set to 10 and the maximum number of iterations to check the interval feasibility constraint for a bead $MNC$ (see Figure 4.7) was set to 100. The results of these measurements per hardware architecture are depicted in Table 6.4 to Table 6.6. A first observation is that when parameter $PET$ is small, $PWF = 100\%$ and $AWB \approx 0$. In this case, the absolute timing constraints are not strict and there is a large number of feasible window assignments. Therefore, such a window assignment can easily be found, almost without backtracking.

<table>
<thead>
<tr>
<th>$MINI$</th>
<th>$PC$</th>
<th>$PET$</th>
<th>$PAR$</th>
<th>10</th>
<th>15</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>60</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.01</td>
<td>0.04</td>
<td>0.03</td>
<td>0.06</td>
<td>0.15</td>
<td>0.09</td>
<td>0.36</td>
<td>0.24</td>
<td>0.25</td>
<td>0.35</td>
</tr>
<tr>
<td>500</td>
<td>75</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
<td>100</td>
<td>100</td>
<td>84</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.01</td>
<td>0.04</td>
<td>0.03</td>
<td>0.10</td>
<td>0.17</td>
<td>0.11</td>
<td>0.21</td>
<td>1.63</td>
<td>1.63</td>
<td>0.38</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>86</td>
<td>92</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.01</td>
<td>0.06</td>
<td>0.03</td>
<td>0.11</td>
<td>0.15</td>
<td>0.14</td>
<td>0.32</td>
<td>2.94</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>50</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.08</td>
<td>0.13</td>
<td>0.10</td>
<td>1.57</td>
<td>0.28</td>
<td>0.24</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>75</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
<td>100</td>
<td>88</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.01</td>
<td>0.04</td>
<td>0.04</td>
<td>0.11</td>
<td>0.23</td>
<td>0.14</td>
<td>1.76</td>
<td>0.44</td>
<td>0.24</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>100</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>92</td>
<td>88</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.02</td>
<td>0.05</td>
<td>0.04</td>
<td>0.18</td>
<td>0.23</td>
<td>0.14</td>
<td>6.78</td>
<td>0.38</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.01</td>
<td>0.04</td>
<td>0.03</td>
<td>0.11</td>
<td>0.16</td>
<td>0.14</td>
<td>14.40</td>
<td>0.78</td>
<td>0.46</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.14</td>
<td>0.25</td>
<td>0.23</td>
<td>0.35</td>
<td>0.45</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>PWF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$AWB$</td>
<td>0.03</td>
<td>0.05</td>
<td>0.04</td>
<td>0.17</td>
<td>0.25</td>
<td>0.22</td>
<td>0.78</td>
<td>0.84</td>
<td>0.49</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Results of performance measurements of the window assignment algorithm for a hardware architecture with NPR = 5.

When parameter $PET$ equals 50%, the algorithm performs very well in terms of $PWF$ for about half of the tested combinations and well for the other ones. In addition, the performance in terms of $AWB$ is still very good. Furthermore, for this value of $PET$, it is hard to identify any regular pattern in the performance, i.e., a systematic relation between the performance and the parameters. This is also the case when the number of processors $NPR$
Table 6.5: Results of performance measurements of the window assignment algorithm for a hardware architecture with NPR = 10.

<table>
<thead>
<tr>
<th>M/C/E</th>
<th>PC</th>
<th>P/E/T</th>
<th>10</th>
<th>30</th>
<th>50</th>
<th>70</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>50</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.51</td>
<td>0.04</td>
<td>0.04</td>
<td>0.59</td>
<td>0.17</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.01</td>
<td>0.04</td>
<td>0.06</td>
<td>0.11</td>
<td>0.18</td>
<td>0.15</td>
</tr>
<tr>
<td>500</td>
<td>75</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.15</td>
<td>0.23</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.15</td>
<td>0.23</td>
<td>0.18</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.15</td>
<td>0.23</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.15</td>
<td>0.23</td>
<td>0.18</td>
</tr>
<tr>
<td>750</td>
<td>50</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td>750</td>
<td>75</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td>750</td>
<td>100</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td>1000</td>
<td>75</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>PW/P</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/H</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AW/B</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
<td>0.12</td>
<td>0.18</td>
<td>0.21</td>
</tr>
</tbody>
</table>

Table 6.6: Results of performance measurements of the window assignment algorithm for a hardware architecture with NPR = 15.

The most difficult combinations are those where P/E/T equals 100%. In this case, the absolute timing constraints are very strict. An interesting pattern that can be recognized is that the PW/F value increases when parameter PAR increases. Thus, the more additional relative timing constraints,
the better the performance. This seems to be paradoxical. However, note that the application generator generates applications such that a feasible bead schedule and thus a feasible window assignment exists. When \( PAR \) increases, the number of relative timing constraints increases. Thus, the number of feasible window assignments decreases but is always larger than zero. Now, it might be the case that when the number of feasible window assignments is small the number of possible windows is small. In that case, a feasible window assignment can be found with a small amount of backtracking.

Another interesting pattern for \( PET = 100\% \) is that the performance in terms of \( PWF \) decreases when parameter \( MINE \) decreases or the number of processors increases. Note that, in both cases, the number of beads increases. Thus, it is likely that the assignment of a window to a bead has influence on a larger number of other beads. Therefore, probably more window shifts are necessary to satisfy the feasibility conditions and thus the amount of backtracking is larger. Since the amount of backtracking is limited by a constant, the algorithm is terminated more often and the performance decreases. Finally, it is hard to distinguish any regular pattern in the \( AWB \) values for \( PET = 100\% \). However, the performance in terms of \( AWB \) is still good.

### 6.4 Local Scheduling Performance

The local scheduling problem is a well-known problem on its own. Therefore, in order to show the usefulness of the local scheduling algorithm in isolation, it was tested with an exhaustive set of problem instances. These instances were generated by a separate bead set generator that can vary all characteristics of the local scheduling problem.

The bead set generator generates a number of bead sets whose characteristics depend on four parameters. Parameter \( AET (\geq 1) \) is the average execution time of the beads, parameter \( DAET (0\% - 100\%) \) is the deviation of the average execution time, parameter \( ANB (\geq 1) \) is the average number of beads per bead set and parameter \( MRL (\geq 1) \) is the maximum relative laxity over the beads of all generated bead sets. The relative laxity of a bead \( b \) is the ratio of the width of the window of the bead and the execution time of the bead: \( (b.wc - b.ws)/b.e \).

The bead set generator generates a bead set by constructing a connected sequence of beads for an interval of length \( AET \times ANB \). Thus, the first
bead of the sequence starts at time 0 and each other bead starts at the end time of its predecessor. The last bead starts before time $AET \times ANB$ and ends at or after time $AET \times ANB$. The execution time $b.e$ of each bead $b$ in the sequence is a random value between $AET - (AET \times DAET/100)$ and $AET + (AET \times DAET/100)$. A relative laxity value between 1 and $MRL$ is randomly generated for each bead. The width of the window $ww$ for each bead is set to $b.e$ times this relative laxity value. Then, the window is randomly placed around the position of bead $b$ in the generated sequence. Thus, if $b$ has start time $st$ in the generated sequence, $b.ws$ is a random value between $st + b.e - ww$ and $st$, and $b.we = b.ws + ww$. By changing the value of $MRL$, the processor utilization can be varied. Note that there is a feasible local schedule for the generated bead set if block sets are generated this way. In addition, all characteristics of a bead set can be varied with these four parameters, and thus the algorithm can be tested with a representative set of randomly generated problem instances.

The performance measures of the algorithm are (1) the percentage $PLF$ of bead sets for which the algorithm can find a feasible local schedule and (2) the average amount of backtracking $ALB$ over all bead sets for which the algorithm can find a feasible local schedule. The amount of backtracking for a single bead set is the number of backtracking steps divided by the number of beads in the set.

The local scheduling algorithm was tested for various combinations of the four parameters of the bead set generator. Since it is not practical to perform tests for each possible combination of values of the four parameters, the number of combinations was restricted. The average execution time $AET$ was restricted to the values 10, 50 and 100. The deviation of the average execution time $DAET$ was restricted to 10%, 50% and 100%. The maximum relative laxity $MRL$ was restricted to values of the set $\{2, 4, ..., 20\}$ and the average number of beads per bead set $ANB$ was restricted to the values 10, 100 and 200. For each combination of these four parameters, 100 different bead sets were generated and used as input for the local scheduling algorithm to determine the performance measures. Performance measurements were carried out on an IBM PC-486 using a Turbo-Pascal 5.0 implementation of the algorithm. For the experiments, the maximum number of bead scheduling attempts $MNRSA$ was set to 100 times the number of beads in a set. This was done because the number of steps needed to find a feasible local schedule for a set of beads is probably exponential to the size of the set.

The bead sets for parameter settings in which $ANB = 200$ and $DAET =$
100% turned out to be the most difficult to schedule. The results of these worst case measurements are depicted in Figure 6.1. For small and large

![Graph showing performance measurements for ANB = 290 and DAET = 100% with varying values of MRL and AET.]

Figure 6.1: Results of performance measurements for ANB = 290 and DAET = 100% with varying values of MRL and AET.

values of parameter MRL, PLF is almost 100% and ALB is near to 0. In addition, for MRL between 12 and 16, the PLF curve shows a minimum and the ALB curve a maximum. This behaviour is characteristic for each parameter setting, whereby the exact values of the minimum and maximum depends on the parameters. The results can be explained as follows. When MRL is small, the number of feasible local schedules is small. Thus, the number of candidates is small and the percentage of candidate beads that are feasibility preserving is large. When MRL increases, the number of feasible local schedules increases, but the number of candidates beads increases even faster, and thus the percentage of candidate beads that are feasibility preserving decreases. When MRL becomes very large, almost every local schedule is feasible and, although the number of candidates is large, almost every candidate bead is feasibility preserving. Apparently, this happens when the value of MRL gets close to 20.

As can be seen in Figure 6.1, an increase in parameter AET results in a decrease of the value of the minimum of the PLF curve and an increase in the value of the maximum of the ALB curve. This behaviour is characteristic for most parameter settings, although there are some exceptions. However, the effect of a change in parameter AET is not very significant compared to other parameters. This was to be expected, because the problem is independent
of the time scale.

As can be seen in Figure 6.2, an increase in the deviation of the average execution time $DAET$ results in a more significant effect on the $PLF$ curve and the $ALB$ curve. An increase of $DAET$ from 10% to 50% results in a moderate decrease of the $PLF$ minimum and a moderate increase of the $ALB$ maximum. Moreover, a further increase of $DAET$ to 100% results in a relatively strong decrease of the $PLF$ minimum and a relatively strong increase in the $ALB$ maximum. This might be explained as follows. When $DAET$ is large, the bead sets consist of beads with a wide variety of execution times. Since the window size of a bead is proportional to the execution time of the bead, these beads also have a wide variety of window sizes. Therefore, there is more variety in nesting of windows. Thus, it is likely that the scheduling of a bead has a larger effect on the schedulability of other beads and the performance of the algorithm decreases.

As can be seen in Figure 6.3, an increase in the average number of beads $ANB$ has a similar effect on the $PLF$ curve and the $ALB$ curve as the parameter $DAET$. An interesting observation is that for each parameter setting in which $ANB$ equals 10, $PLF = 100\%$ and $ALB = 0$. Thus, for small bead sets, the algorithm always finds a feasible local schedule without backtracking. However, large bead sets are apparently more difficult to schedule. This might be explained as follows. Obviously, the scheduling of a bead has an effect on the schedulability of other beads. Thus, if there are
more beads to be scheduled, it is likely that in general the effects on the schedulability of other beads will add up.

The worst measured value for PFBS is 94% for parameter setting $ANB = 200$, $DAET = 100\%$, $AET = 100$ and $MRL = 16$ as depicted in Figure 6.1. However, although it is not depicted in the figure, the results show that for 66% of the bead sets of this parameter setting, a feasible schedule is found in a number of steps that is at most twice the size of the set to be scheduled.

Finally, the computation time of the algorithm seems to be quadratic in the number of beads to be scheduled. The computation time for $ANB = 10$ is in the order of 50 milliseconds. The average computation time for $ANB = 100$ is approximately in the order of 3 seconds and for $ANB = 200$ in the order of 30 seconds with best cases of 500 milliseconds and worst cases of 6 minutes. Thus, the average computation time is far less than the worst-case computation time that is determined by the time-complexity $O(|PB|^3)$ of the algorithm.

6.5 Entire Approach Performance

The performance measures of the entire scheduling approach are (1) the percentage $PEF$ of applications for which a correct process assignment and a feasible bead schedule is found, and (2) the average amount of backtrack-
ing $AEB$ that was necessary to find these assignments and schedules. The amount of backtracking for a single application is expressed as the sum of the amounts of backtracking for each step of the approach.

The entire approach was tested for the same three hardware architectures for which the process and window assignment algorithms were tested. Also the values of the parameters of the application generator were the same as for the tests of these algorithms. Per generated architecture, 25 applications were generated for each combination of these parameters. Then, for each set of 25 applications, the two performance measures of the entire approach were determined by running the process assignment algorithm, the window assignment algorithm and the local scheduling algorithm successively. The values of the parameters of each algorithm were identical as for the tests of these algorithms in isolation. The results of these measurements per hardware architecture are depicted in Table 6.7 to Table 6.9. The performance

<table>
<thead>
<tr>
<th>$MINK$</th>
<th>$NC$</th>
<th>$PET$</th>
<th>$50$</th>
<th>$75$</th>
<th>$100$</th>
<th>$50$</th>
<th>$75$</th>
<th>$100$</th>
<th>$50$</th>
<th>$75$</th>
<th>$100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$500$</td>
<td>$50$</td>
<td>$7E9$</td>
<td>$6E9$</td>
<td>$8E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
</tr>
<tr>
<td>$750$</td>
<td>$75$</td>
<td>$7E9$</td>
<td>$6E9$</td>
<td>$8E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
<td>$6E9$</td>
</tr>
<tr>
<td>$1000$</td>
<td>$100$</td>
<td>$PET$</td>
<td>$50$</td>
<td>$75$</td>
<td>$100$</td>
<td>$50$</td>
<td>$75$</td>
<td>$100$</td>
<td>$50$</td>
<td>$75$</td>
<td>$100$</td>
</tr>
</tbody>
</table>

Table 6.7: Results of performance measurements of the complete approach for a hardware architecture with $NPR = 5$.

of the process assignment algorithm in the entire approach is identical to the performance of the algorithm in isolation, because the same applications were used. These performance measures can be found in Tables 6.1 to 6.3.

A first observation that can be made is that, although it is not shown in the tables, each window assignment found in the entire approach is feasible and a local schedule can be constructed for each processor without backtracking. This result supports the usefulness of decoupling the local scheduling problems by using the window approach. It also implies that the combina-
6.5. Entire Approach Performance

<table>
<thead>
<tr>
<th>MINE</th>
<th>PC</th>
<th>PEF</th>
<th>PAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>60</td>
<td>PEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEB</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.03</td>
<td>0.05</td>
<td>0.02</td>
<td>0.07</td>
</tr>
<tr>
<td>500</td>
<td>75</td>
<td>PEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEB</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
<td>0.10</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>PEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEB</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.13</td>
</tr>
<tr>
<td>250</td>
<td>50</td>
<td>PEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEB</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.12</td>
</tr>
<tr>
<td>250</td>
<td>75</td>
<td>PEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEB</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.02</td>
<td>0.02</td>
<td>0.03</td>
<td>0.10</td>
</tr>
<tr>
<td>250</td>
<td>100</td>
<td>PEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AEB</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.02</td>
<td>0.03</td>
<td>0.03</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 6.8: Results of performance measurements of the complete approach for a hardware architecture with NPR = 10.

This paragraph, the performance of the entire approach is discussed in isolation. In the next paragraph, this performance is compared with the performance of the window assignment algorithm in isolation. It can be seen in Tables 6.7 to 6.9 that the amount of backtracking of the entire approach is small, i.e., the performance in terms of AEB varies between zero and one. Thus, the number of backtracking steps is at most the number of beads. Therefore, at least for the tested cases, the goal of the window approach to keep the amount of backtracking small, as discussed in Chapter 2, is achieved. Furthermore, it is again hard to recognize any regular pattern in the AEB values. On the other hand, the performance in terms of PEF shows more regular patterns. The PEF value typically decreases when PET or PAR increases. In this case, the absolute timing constraints become more strict and the number of additional relative timing constraints increases respectively. In addition, PEF also typically decreases when parameter NPR increases or parameter MINE decreases. This could be expected, because
### Table 6.9: Results of performance measurements of the complete approach for a hardware architecture with NPR = 15.

In this case the number of beads increases. Note that a combination of such changes in these four parameters has an even larger effect on PEF. Finally, no regular pattern can be recognized in the PEF values when parameter PC is varied. One would expect that by increasing PC, the performance decreases because there are more consistency constraints due to more communication. An acceptable explanation for the behavior of parameter PEF in this case has not been found.

Tables 6.1 to 6.3 show that the process assignment algorithm almost always finds a correct process assignment without backtracking. As mentioned before in this section, in the entire approach a local schedule can be constructed for each processor without backtracking. From these two observations, it can be concluded that the performance of the entire approach is determined by the performance of the window assignment algorithm. Thus, the performance of the entire approach can be compared to the performance of the window assignment algorithm in isolation. The latter performance measurements are depicted in Tables 6.4 to 6.6. It can be seen that the PEF values for the complete approach are typically smaller than the corresponding PWF values for the window assignment algorithm in isolation, except when PAR = 0% and PFT = 100%. The decrease is obviously caused by the process assignment algorithm. Although a process assignment found is correct, this does not mean that it is also feasible. In addition, it might be harder to find a window assignment for the process assignments found.
as for the assigned applications used for testing the window assignment algorithm in isolation. An acceptable explanation for the exceptional case, where $PAR = 0\%$ and $PET = 100\%$ has not been found.

An overall conclusion is that the entire approach performs very well as long as the absolute timing constraints are not too strict, i.e., the processor utilization is not near 100\% or the number of beads does not exceed approximately 500. However, the performance is still acceptable for applications with approximately 250 beads and a high processor utilization, for instance for $NPR = 10$, $PET = 100\%$, $MINE = 250$ and $PAR = 0\%$.

### 6.6 Conclusion

In this chapter, results of performance measurements of each algorithm in isolation and of the entire approach are presented. These measurements were carried out using input from an architecture generator and an application generator. The application generator generates applications for which there is a process assignment for which a feasible bead schedule exists. The performance measurements show that the local scheduling algorithm is quite efficient with respect to the percentage of problem instances for which a solution is found. In addition, the computation time on a standard PC needed to find a solution is acceptable even for large instances. The process assignment algorithm in isolation performs very well. In addition, the combination of the two feasibility conditions and the shifting techniques used in the window assignment algorithm turned out to very effective. In particular, each window assignment found by the algorithm is feasible and a local schedule can be found for each processor without backtracking. The performance of the entire approach is very good as long as the processor utilization is not near 100\% or the number of beads is exceeds 500. However, the performance is still acceptable for applications with approximately 250 beads and a high processor utilization.
Chapter 7

Conclusions

In this thesis, a novel realistic pre-run-time scheduling model is presented. The model is formalized to facilitate the presentation and verification of the resulting pre-run-time scheduling problems and algorithms. The basic scheduling units defined by the model are pieces of code, called beads, between successive pre-defined preemption points. This novel notion of semi-preemptability provides a trade-off between the two extremes of preemption at any point and non-preemptability. The model also incorporates relative timing constraints for the synchronization of concurrent activities which have not yet been considered in the context of pre-run-time scheduling.

The pre-run-time scheduling approach first attacks a process assignment problem not yet described in the literature. An algorithm is proposed that uses a novel objective function that weights communication against parallelism. The approach also utilizes a novel window technique to solve the scheduling problem. Thereby, an execution interval is assigned to each bead such that all constraints are satisfied when each bead is scheduled in its window. The main advantage of the window approach is that the scheduling problem at each processor can be solved in isolation. In addition, the window approach leads to only a small amount of backtracking. This is supported by performance measurements presented in Chapter 6. Finally, a new scheduling algorithm is presented for the well-known single processor scheduling problem with release times and deadlines.

Results of performance measurements of the three steps of the scheduling
approach in isolation and of the entire approach are presented and discussed in Chapter 6. The local scheduling problem is a well-known problem on its own. Therefore, in order to show the usefulness of the local scheduling algorithm in isolation, it was tested with an exhaustive set of problem instances. These instances were generated by a separate bead set generator that can vary all characteristics of the local scheduling problem. The bead sets contained up to 200 beads and the processor utilization ranged from 10 to 100%. The results show that the local scheduling algorithm finds a feasible local schedule in almost all cases. In addition, the average computation time of the algorithm is quadratic in the number of beads. The process assignment algorithm, the window assignment algorithm and the entire approach were tested using input from a hardware architecture generator as well as an application generator. The generators generate architectures and applications according to the model presented in this thesis such that a feasible assignment and schedule exists. The performance of the process assignment algorithm in isolation is very good. In addition, the combination of the two feasibility conditions and the shifting techniques used in the window assignment algorithm turned out to very effective. In particular, each window assignment found by the algorithm is feasible and a local schedule can be subsequently constructed for each processor without backtracking. The performance of the entire approach is very good as long as the absolute timing constraints are not too strict, i.e., the processor utilization is not near to 100%, or the number of beads is not too large, i.e., more than 500 beads. Moreover, the performance is still acceptable for applications with approximately 250 beads and a high processor utilization.

Future research includes the extension of the pre-run-time scheduling model to accommodate static fault-tolerance by replication of hardware facilities, processes and activities. In addition, a complete testbed should be implemented. This testbed supports more comprehensive testing of the algorithms presented in this thesis and the development of new heuristics. In addition, other types of scheduling algorithms, such as simulated annealing and genetic algorithms, can be developed using the testbed. Also, the pre-run-time scheduling approach has to be tested with practical applications. Currently, a design of a copier and a television set are worked out in detail that can be used for this purpose.

Another interesting topic for future research is the optimization of the application in order to increase the probability that a feasible schedule is found. For instance, techniques that exploit inherent parallelism can be used, such as cloning of frequently used facilities in conjunction with the
introduction of asynchronous remote procedure calls. Then, an infeasible schedule is repeatedly transformed in order to come to a feasible one, by cloning and reassigning of facilities and objects. Thereby, the assignment and the local schedules are adjusted accordingly and the complete construction of a new schedule is avoided. This technique is currently under development at the New Jersey Institute of Technology and preliminary results can be found in [52].

Finally, the pre-run-time scheduling approach can also be optimized in order to significantly reduce the complexity of the problem. One way is to group beads at the same processor into larger blocks that are used in the subsequent scheduling problem as non-preemptable scheduling units. Thereby, the knowledge that, due to logical dependencies, only certain combinations of program branches can occur can be used. This technique is called linking and is described in [40]. In addition, the fact that, due to data-dependency, only one of the alternatives of a conditional will be executed at run-time can be exploited. Thus, beads of different alternatives of the same conditional are grouped with overlap in time into a single block. Also, hierarchical scheduling is a candidate for future research. Such a technique is useful in conjunction with the design of a real-time system in which the real-time constraints and a corresponding schedule are repeatedly refined [12]. The advantage of this approach is that the total scheduling problem is split into smaller pieces that can be solved in reasonable time although they are NP-complete.
Bibliography


<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>a path of beads connected by precedence relations</td>
</tr>
<tr>
<td>••</td>
<td>a subgraph of beads connected by precedence relations</td>
</tr>
<tr>
<td>⊕</td>
<td>the concatenation operator for two local schedules</td>
</tr>
<tr>
<td>∅</td>
<td>the empty set</td>
</tr>
<tr>
<td>∞</td>
<td>infinity</td>
</tr>
<tr>
<td>a_{ds,ds}</td>
<td>the blocking time of an access to dv that involves data of size ds</td>
</tr>
<tr>
<td>δ_{cm,ds}</td>
<td>the communication delay for data of size ds via cm</td>
</tr>
<tr>
<td>ω_c</td>
<td>communication objective weight</td>
</tr>
<tr>
<td>ω_p</td>
<td>parallelism objective weight</td>
</tr>
<tr>
<td>AB(pr)</td>
<td>the set of beads assigned to processor pr</td>
</tr>
<tr>
<td>ADO</td>
<td>Abstract Data Object</td>
</tr>
<tr>
<td>ADT</td>
<td>Abstract Data Type</td>
</tr>
<tr>
<td>AEB</td>
<td>average entire approach backtracking</td>
</tr>
<tr>
<td>AET</td>
<td>average bead execution time</td>
</tr>
<tr>
<td>ALB</td>
<td>average local scheduling backtracking</td>
</tr>
<tr>
<td>ANB</td>
<td>average number of beads</td>
</tr>
<tr>
<td>APB</td>
<td>average process assignment backtracking</td>
</tr>
<tr>
<td>APS(P,A)</td>
<td>the set of assigned processes</td>
</tr>
<tr>
<td>APS(pr,P,A)</td>
<td>the set of processes assigned to processor pr</td>
</tr>
<tr>
<td>AWB</td>
<td>average window assignment backtracking</td>
</tr>
<tr>
<td>b, b_i</td>
<td>a bead</td>
</tr>
<tr>
<td>b, ac</td>
<td>the activity to which b belongs</td>
</tr>
<tr>
<td>b, cm</td>
<td>the communication medium accessed by b</td>
</tr>
</tbody>
</table>
b.d the deadline of b
b.ddp the data-dependency probability of b
b.ds the size of data to be received at the start of b
b.e the execution time of b
b.est the earliest start time of b
b.levwc the largest correct window end time for b
b.levws the largest correct window start time for b
b.lw the largest window width for b
b.pr the processor to which b is assigned
b.ps the process in which b executes
b.scwce the smallest correct window end time for b
b.scws the smallest correct window start time for b
b.st the start time of b
b.wc the window end time of b
b.ws the window start time of b
B the set of beads
BS a bead schedule
BS(b) the start time of b
CC the set of consistency constraints among subgraphs of beads in B
cm, cm_i a communication medium
cm.bd the communication delay for bandwidth size data via cm
cm.bw the bandwidth of communication medium cm
cm.mod the communication mode of cm
CM the set of communication media
DAET deviation in the average bead execution time
DAG Directed Acyclic Graph
dv a device
dv.cb the constant blocking time for device dv
dv vb the variable blocking time for device dv
DV the set of devices
c_i = b_i.e
Glossary

\( epx(b, t) \) the probability that bead \( b \) executes at time \( t \)
\( et(\text{LS}(SB)) \) the end time of local schedule \( \text{LS}(SB) \)
\( cu(PB, t) \) the expected utilization for the set of beads \( PB \) at time \( t \)
LAN Local Area Network
LCM Least Common Multiple
\( lo \) the largest offset of a relative timing constraint
LPC Local Procedure Call
\( LS(PB) \) a local schedule for the beads in \( PB \)
\( MINE \) minimum execution time of a generated bead
\( MNA \) maximum number attempts to assign a process
\( MNBSA \) maximum number of bead scheduling attempts
\( MNC \) maximum number of checks for the interval feasibility constraint
\( MNI \) maximum number of adjustment iterations
\( N \) the natural numbers (including 0)
\( N^+ \) the positive natural numbers
\( NCPR \) maximum number of candidate processors
\( NPR \) number of processors
\( MRL \) maximum relative bead laxity
\( O \) the time-complexity order function
\( PA \) a process assignment
\( PA(ps) \) the processor to which process \( ps \) is assigned
\( PAR \) percentage relative timing constraints in a generated application
\( PB, PB_i \) a set of beads assigned to the same processor
\( PC \) percentage calls in a generated application
\( PEF \) percentage entire approach solutions found
\( PET \) percentage execution time of a generated bead
\( PLF \) percentage local schedules found
\( PPF \) percentage process assignments found
\( pr, pr_i \) a processor
\( pr.md \) the maximum number of devices that can be connected to
processor $pr$

$PR$ the set of processors

$ps$ a process

$ps.e$ the execution time of process $ps$

$ps.nd$ the number of devices managed by process $ps$

$PS$ the set of processes

$PWF$ percentage window assignments found

$r_{o_{i,j}}$ the route of communication media from processor $pr_i$ to processor $pr_j$

$RO$ the set of routes

$RPC$ Remote Procedure Call

$RTC$ the set of relative timing constraints among beads in $B$

$so$ the smallest offset of a relative timing constraint

$st_i = b_i.st$

$st(b, LS(B))$ the start time of $b$ after local schedule $LS(B)$

$t$ a point in time

$WA$ a window assignment

$WA(b).wc$ the window end time of $b$

$WA(b).ws$ the window start time of $b$

$WAN$ Wide Area Network

$we_i = b_i.wc$

$ws_i = b_i.ws$
Property Index

AbsoluteTimedCorrectIntervals ..................................................... 82
AbsoluteTimingConstraintsAssured ............................................. 41
AbsoluteTimingConstraintsSatisfied ........................................... 35
BeadsInWindows ..................................................................... 40
ConnectedSequence ................................................................... 113
ConsistencyConstraintsAssured .................................................. 43
ConsistencyConstraintsSatisfied .................................................. 36
CorrectProcessAssignment ............................................................ 33
CorrectProcessor ....................................................................... 32
CorrectWindowAssignment ............................................................. 44
FeasibilityPreservingBead ............................................................. 109
FeasibilityPreservingBeadSequence ............................................. 112
FeasibleBeadSchedule ................................................................. 36
FeasibleCorrectIntervalPairs ....................................................... 99
FeasibleLocalSchedule ................................................................. 49
FeasiblePairs ........................................................................... 46
FeasibleSets ............................................................................. 46
IndependentPartialLocalSchedule .............................................. 116
MinimumCorrectIntervalDistance ............................................... 82
MinimumWindowWidth ................................................................. 40
<table>
<thead>
<tr>
<th>Property Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextPreferredBead</td>
<td>129</td>
</tr>
<tr>
<td>NoOverlap</td>
<td>35</td>
</tr>
<tr>
<td>PrefixFeasibleLocalSchedule</td>
<td>109</td>
</tr>
<tr>
<td>PresumablyFeasibleWindowAssignment</td>
<td>48</td>
</tr>
<tr>
<td>ProcessorNotOverloaded</td>
<td>74</td>
</tr>
<tr>
<td>ProgressMakingBeads</td>
<td>128</td>
</tr>
<tr>
<td>RelativeTimedCorrectIntervals</td>
<td>83</td>
</tr>
<tr>
<td>RelativeTimingConstraintsAssured</td>
<td>42</td>
</tr>
<tr>
<td>RelativeTimingConstraintsSatisfied</td>
<td>36</td>
</tr>
<tr>
<td>WindowEqualsCorrectIntervals</td>
<td>84</td>
</tr>
</tbody>
</table>
Index

B, 26
CC, 31
CM, 10
DV, 11
PR, 9
PS, 19
RO, 11
RTC, 30

absolute timing constraint, 29
activity, 20, 26
ADO, 14
ADT, 12
application, 16
application generator, 135
assigned bead to processor set, 33
assigned process set, 32
assigned process to processor set, 32
asynchronous communication, 21
backtracking, 125
bandwidth communication delay, 11
bead, 25
bead execution interval, 55
bead schedule, 35
bead scheduling problem, 37
bead selection, 94, 122
bead set generator, 143
blocking device access, 23
blocking time, 24

body, 16
bus, 10

candidate processor set construction, 71
candidate set construction, 121
class, 12
communication amount, 63
communication delay, 22
communication deletion, 73
communication insertion, 34, 65
communication media interconnection topology, 11
communication medium, 10, 28
communication medium bandwidth, 11
communication mode, 21
concatenate operator, 108
conditional, 15
consistency, 31
consistency constraint, 31
consistency constraint transformation, 91
constant blocking time, 23
correct interval, 81
correct interval adjustment, 85, 99
correctness constraints, 85
data, 14
data-dependency, 28
data-dependency probability, 28
deadline, 19, 20, 29
delimited, 25
dependability, 1
deterministic pre-run-time view, 25
device configuration, 11
earliest deadline first scheduling, 2
earliest start time, 29
entire approach performance measures, 147
execution, 20
execution DAG, 30
execution path, 20
execution time, 26
expected bead parallelism, 61
expected execution probability, 60
expected processor parallelism, 62
expected utilization, 60
export, 17
facility, 13
feasibility condition, 45, 70
feasibility condition check, 103
feasible window assignment, 45
hardware architecture, 9
hardware architecture generator, 134
heuristic algorithm, 7
hop, 11
import, 17
instance, 29
largest offset, 30
largest window width, 84
latest backtracking pair, 129
LCM, 26
link, 10
local schedule, 49
local scheduling algorithm, 119
local scheduling performance measures, 144
local scheduling problem, 50
LPC, 20
maximum processor devices, 12
medium communication amount, 63
method, 12
network, 10
non-blocking device access, 23
NP-complete, 4
object, 14
object-based programming paradigm, 12
parameter data, 20
parameter data size, 28
path, 30
period, 19, 20
physical devices, 11
pre-run-time scheduling, 3
pre-run-time scheduling approach, 37
precedence relation, 26
preemption point, 24
private memory, 10
process, 16, 26
process assignment, 32
process assignment algorithm, 64
process assignment performance measures, 138
process assignment problem, 33
process devices, 19
process execution time, 67
process reassignment, 74
process selection, 67
processor, 9, 33
processor selection, 68
Index

processor selection objective function, 70

window shifting, 102
window start time, 40

rate-monotonic scheduling, 2
recursion, 15
relative timing constraint, 29
repetition, 15
replication, 15
route, 11
RPC, 21
run-time scheduling, 1

schedulability analysis, 2
scheduling interval, 26
selection probability, 15
semi-preemptable, 24
shared memory module, 11
smallest offset, 30
start time, 28
statement, 15
subgraph, 31
synchronous communication, 22
system properties, 54

time-complexity, 4
timeliness, 1
timing constraints, 19
type, 12

variable, 13
variable blocking time, 23

window, 39
window assignment, 40, 95
window assignment algorithm, 80, 89
window assignment performance measures, 140
window assignment problem, 48
window end time, 40
Samenvatting

Het garanderen van tijdigheid in complexe gedistribueerde real-time syste- 
men is een bekend probleem. De omgeving van zo een systeem kan bepaalde 
tijdseisen opleggen aan de taken van het systeem. Deze tijdseisen moeten 
gehaald worden om catastrofale gevolgen voor de omgeving te voorkomen. 
Een bekende methode om tijdigheid te garanderen is pre-run-time schedu- 
ling. Bij deze methode wordt vooraf een schedule bepaald zodanig dat alle 
tijdseisen worden gehaald. Pre-run-time scheduling is bijzonder nuttig voor 
apPLICATIES waarvan alle karakteristieken van te voren bekend zijn, zoals in-
dustriële automatisering, robotica, telecommunicatie en luchtvaart.

In deze dissertatie wordt een nieuw pre-run-time scheduling model ge-
presenteerd voor gedistribueerde real-time systemen. De hardware architec-
tuur van zo een systeem bestaat uit een verzameling processoren verbonden 
doorn een communicatie netwerk en een aantal sensoren en actuatoren. Een 
apPLICatie bestaat uit een verzameling objecten die geclustered zijn in pro-
cessen. De processen definiëren een verzameling parallele executies. Een 
van de nieuwe eigenschappen van het model is dat zulk een executie alleen 
onderbroken kan worden op van te voren vastgelegde punten, zoals bijvoor-
beeld een aanroep van een procedure in een ander process. De dynamische 
structuur van een applicatie wordt daarom gemodelleerd als een graaf van 
ononderbroken stukjes programma code tussen twee opeenvolgende onder-
brekingspunten. Zo'n stukje code wordt een bead genoemd. Het model 
bevat verschillende soorten eisen waaraan de beads van een applicatie moe-
ten voldoen. Absolute tijdseisen reprenten periodiciteit en deadlines van 
execties, relatieve tijdseisen modelleren verschillende vormen van voorgan-
gers relaties tussen beads, en consistentie eisen dwingen resource consistentie 
af. Nog niet eerder zijn relatieve tijdseisen beschouwd tijdens pre-run-time 
scheduling.

Gebaseerd op het model wordt een pre-run-time scheduling aanpak voor-
gesteld waarmee de scheduling problemen kunnen worden opgelost. Deze
aanpak bestaat uit drie stappen. Ten eerste worden alle processen van een applicatie toegewezen aan de processoren van het gedistribueerde systeem. In de tweede en de derde stap wordt er een start tijd toegekend aan iedere bead. Daarvoor wordt er eerst een executie interval, ook wel window genaamd, toegewezen aan iedere bead. Dit gebeurt zodanig dat aan alle opgelegde eisen is voldaan wanneer iedere bead executeert in zijn window. In de derde stap wordt aan iedere bead per processor een start tijd toegekend zodanig dat iedere bead executeert in zijn window en geen twee beads op dezelfde processor overlappen in de tijd. Het voordeel van deze nieuwe window methode is dat het probleem in de derde stap onafhankelijk per processor kan worden opgelost. Tevens wordt het aantal stappen dat nodig is om een oplossing te vinden klein gehouden. Dit blijkt uit test resultaten gepresenteerd in Hoofdstuk 6.

Drie constructieve heuristische algoritmen worden gepresenteerd om de problemen van iedere stap op te lossen. Een heuristisch algoritme gebruikt probleem-georiënteerde regels die eenvoudig te evalueren zijn zodat een niet-optimale oplossing gevonden kan worden binnen redelijke tijd. Het process toewijzings algoritme gebruikt een nieuwe doelfunctie die communicatie en parallelisme tegen elkaar afwegt. Het window toewijzings algoritme probeert een window toewijzing te vinden die voldoet aan twee zogeheten feasibility condities. Daarvoor gebruikt het algoritme een techniek die windows verschuift zodanig dat aan beide condities is voldaan. Het lokale scheduling algoritme vindt een lokaal schedule door een partieel schedule herhaaldelijk aan het eind uit te breiden.

Testresultaten laten zien dat het lokale scheduling algoritme een lokaal schedule vindt voor bijna alle geteste gevallen. Deze gevallen bevatten tot 200 beads en de processor bezetting varieert van 10 tot 100%. Het process toewijzings algoritme op zich gedraagt zich bijzonder goed. Tevens blijkt de combinatie van de twee feasibility condities en de verschuivingstechniek van het window toewijzings algoritme zeer effectief te zijn. Voor iedere gevonden window toewijzing kan een lokaal schedule per processor gevonden worden zonder backtracking. Het gedrag van de gehele aanpak is zeer goed zolang de absolute tijdseisen niet te stric zijn en dus de processor bezetting niet tegen de 100% loopt. Tevens wordt het gedrag minder acceptabel wanneer het aantal beads meer dan ongeveer 500 is. Desondanks gedraagt de scheduling aanpak zich nog steeds acceptabel voor applicaties met 250 beads en een hoge processor bezetting.
Curriculum Vitae

Jack Verhoosel was born on September 26, 1964 in Maastricht. He received his diploma in Computing Science from the Eindhoven University of Technology, The Netherlands, in 1988. The last year of his study he worked on a Parallel Document Retrieval System at the Philips Research Laboratories Eindhoven. Since 1988, he has been with the Department of Mathematics and Computing Science at the Eindhoven University of Technology. During the first year at this university he has done research on formal specification languages for embedded systems. At the end of 1989 he switched to a research area that includes specification, design as well as scheduling of distributed real-time systems. Under supervision of Prof.Dr. D.K. Hammer he carried out the research that has led to this thesis. He has held the position of Visiting Researcher at the New Jersey Institute of Technology, New Jersey, USA. Currently, he is a researcher at the Telematica Research Centrum in Enschede, The Netherlands.
Pre-Run-Time Scheduling of Distributed Real-Time Systems

Models and Algorithms

van

Jack Verhoosel
1. Beschouw een verzameling taken met per taak een hoeveelheid tijd benodigd om de taak uit te voeren, een vroegste tijd waarop de taak kan starten en een deadline waarvoor de taak beëindigd moet zijn. Een algemeen bekend scheduling probleem is het toekennen van een starttijd aan iedere taak zodanig dat (1) het uitvoeren van iedere taak zonder onderbreking plaats vindt tussen zijn vroegste starttijd en deadline en (2) op ieder tijdstip hoogstens één taak wordt uitgevoerd. Ondanks de intuïtieve eenvoud van dit probleem, is het in 1977 reeds bewezen dat het probleem NP-compleet is [1]. Echter, men hoeft niet ontmoeidigd te geraken, daar voor een realistische verzameling taken dit probleem ook opgelost kan worden door het heuristische algoritme met een polynomiale tijdscomplexiteit beschreven in Hoofdstuk 5 van dit proefschrift.


2. Een algoritme lost een probleem op als het een oplossing kan vinden voor iedere probleem instantie waarvoor een oplossing bestaat. Een algoritme dat een NP-compleet probleem oplost wordt optimaal genoemd. Echter, de NP-compleetheid impliceert dat zo een algoritme een exponentiële tijds-complexiteit heeft [1]. Dit betekent dat er altijd probleem instanties zijn waarvoor zo een algoritme geen oplossing kan vinden binnen “acceptabele tijd”. Het praktische nut van de optimaliteit van een algoritme kan dus in twijfel worden getrokken. Voor de algoritmen gepresenteerd in dit proefschrift is vanwege deze reden niet gestreefd naar optimaliteit.


3. Het doel van pre-run-time scheduling voor traditionele systemen zonder real-time constraints is in het algemeen het optimaliseren van een bepaald criterium, zoals bijvoorbeeld het minimaliseren van de schedule lengte [1]. Voor realistische real-time systemen echter mag men al bij zijn wanneer een schedule gevonden wordt dat aan alle real-time constraints voldoet. Optimalisatie van een
bepaald criterium is voor deze systemen dus van secundair belang en maakt daarom geen onderdeel uit van de onderzochte problemen in dit proefschrift.


4. Het vinden van een goed algoritme voor een NP-compleet probleem is een frustrerende bezigheid, daar er altijd probleem instanties zijn waarvoor een oplossing niet gevonden kan worden binnen “acceptabele tijd”. Aangezien de meeste problemen in het leven NP-compleet zijn, zou men kunnen concluderen dat het leven frustrerend is of niet-algoritmisch benaderd moet worden.

5. Het is inmiddels algemeen bekend dat de op den duur dodelijke ziekte AIDS overgebracht kan worden door middel van geslachtsgemeenschap. Daar sommige mensen nog steeds geen condoom gebruiken en veel wisselende sexuele contacten onderscheiden lijkt er een nieuwe vorm van Russisch roulette te zijn ontstaan. Dit is kennelijk een vreemde uiting van het op zich gezonde menselijke streven naar uitdagingen.

6. Aangezien de laatste jaren de treinprijzen relatief meer zijn gestegen dan de kosten van auto rijden lijkt het wel of de overheid het autogebruik wil stimuleren ten koste van het openbaar vervoer.


8. De toenemende agressiviteit van automobilisten tegen wegwerk- kers in verband met de slechte verkeersdoorstroom is moeilijk verklaarbaar als men zich realiseert dat wegwerkzaamheden juist een betere verkeersdoorstroom tot doel hebben. Dit kenmerkt mensen die zich concentreren op de korte termijn en moete hebben vooruit te kijken.

10. Als de stijging in de afgelopen jaren van de wachttijd voor een ziekenhuis operatie geëxtrapoleerd wordt ziet men dat mensen in de toekomst pas na hun dood geopereerd zullen worden.