Resource Budgeting as a Tool for Reduced Development Cost for Embedded Real-time Computer Systems

MATTIAS WECKSTÉN

School of Information Science, Computer and Electrical Engineering
HALMSTAD UNIVERSITY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2004
Resource Budgeting as a Tool for Reduced Development Cost for Embedded Real-time Computer Systems

MATTIAS WECKSTÉN

Technical Report no. 33L
ISSN 1651-4963
Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Contact information:
Mattias Weckstén
School of Information Science, Computer and Electrical Engineering
Halmstad University
Kristian IV:s väg 3
SE–301 18 Halmstad, Sweden
Phone: +46 (0)35–16 73 96
Fax: +46 (0)35–12 03 48
Email: mattias.wecksten@ide.hh.se
URI: http://www.hh.se/staff/weck/

Cover:
Schematic overview of the proposed budgeting tool.

Back:
Written by Jörgen A. Hammarqvist.
Photo by Kristina Weckstén.
Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2004
Resource Budgeting as a Tool for Reduced Development Cost for Embedded Real-time Computer Systems

MATTIAS WECKSTÉN
School of Information Science, Computer and Electrical Engineering,
Halmstad University
Department of Computer Science and Engineering,
Chalmers University of Technology

Thesis for the degree of Licentiate of Engineering, a Swedish degree between M.Sc. and Ph.D.

Abstract

Wouldn’t it be great if there were a systematic method for derivation of non-functional constraints available at design time that made it possible to verify design and make implementation a much clearer task? This kind of methods are needed since systems of increasing complexity has to be developed, and the cost for failing has proven to be too high. The problem is how to derive the design time constraints into implementation time constraints, maintaining the traceability for the individual constraints, and early on get indications whether a project is about to fail or not.

A method for implementation time constraint derivation has been developed and is presented in this thesis. Along with the basic method, several extensions are proposed. Evaluations of the practical usefulness of the method and the method’s scalability have been done. To prove the method’s importance in real development projects, a method for evaluation of the usability of this kind of methods has also been developed. The evaluation of the practicality shows that it is possible to find close to optimal solutions (within percent) in short time (within minutes). The evaluation of the scalability shows that the run time for finding implementable solutions scales polynomial with the size of the task graph. The evaluation of the usability shows that using the proposed method always leads to lower development cost than using an ad hoc method, in the case that the implementation is about to fail.

Keywords: Real-time systems, embedded, resource budgeting, design tool, tightness optimization, guarantees.
Preface

First of all I would like to thank my family and all my friends for their support throughout the years. Thanks to my supervisor, Jonas Vasell, who guided me through the whole process from the first trembling experiments to the last letter of this thesis. Thanks also to Magnus Jonsson, who helped me publish my results and supervised the writing of this thesis. Thanks also to Yu Feng Zhao, without whom, this thesis would not be.

"If you can see the light in the end of tunnel, what is your plan if it turns out to be a train?"
## Contents

1 Introduction .................................................. 1
   1.1 Background ............................................ 1
   1.2 Focus and limitations ................................. 4
   1.3 Contributions ........................................ 6
   1.4 Thesis overview ...................................... 7

2 The BICD Method ........................................... 9
   2.1 Introduction ........................................... 9
   2.2 Problem statement and assumptions .................. 10
   2.3 The BICD method ...................................... 11
      2.3.1 Method ............................................ 12
      2.3.2 Example .......................................... 14
   2.4 Complexity ............................................ 17
   2.5 Use of BICD ........................................... 19

3 Extending the BICD ......................................... 21
   3.1 Introduction ........................................... 21
   3.2 Multiple task instances ............................... 21
   3.3 Periodic systems ...................................... 25
   3.4 Varying quality of the estimates .................... 28
   3.5 Other types of requirements ......................... 28

4 Validation .................................................. 31
   4.1 Introduction ........................................... 31
   4.2 Practicality .......................................... 31
      4.2.1 Method ............................................ 32
      4.2.2 Experiment ....................................... 33
      4.2.3 Discussion ....................................... 38
   4.3 Usefulness ............................................ 39
      4.3.1 Method ............................................ 41
      4.3.2 Experiment ....................................... 42
CONTENTS

4.3.3 Discussion ........................................... 43

5 Conclusions and Future Work ................................... 47

A Related Work ............................................. 51

A.1 Introduction .............................................. 51

A.1.1 Deriving internal timing constraints ....................... 51

A.1.2 A UML-based design methodology for real-time and em-
bedded systems ........................................... 52

A.1.3 Guaranteeing real-time requirements with resource-based
    calibration of periodic processes ........................... 53

A.1.4 Dynamic end-to-end guarantees in distributed real-time
    systems ................................................ 53

A.1.5 The slack method: a new method for static allocation of
    hard real-time tasks ..................................... 54

A.1.6 Resource conscious design of distributed real-time sys-
    tems: an end-to-end approach .............................. 54

A.1.7 Experiences from introducing state-of-the-art real-time tech-
    niques in the automotive industry ......................... 55

B Detailed Case Descriptions ................................... 57

B.1 slackUA .................................................. 60

B.2 baseStationPA ............................................ 62

B.3 baseStationPAC ............................................ 65

B.4 smallUA .................................................. 68

B.5 simpleRegPA .............................................. 69

B.6 neuralNetUA .............................................. 72
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of the suggested development flow model with requirement forwarding. Validation of the design at an early stage makes it possible to avoid future dead ends.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>In <em>ad hoc</em> development methods the non functional constraints will not be used to verify the system until after implementation.</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td><em>Ad hoc</em> development makes use of the non functional requirements at the implementation stage (for validation).</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>In the budgeting methods non functional constraints are used for validation of dimensioning.</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Flowchart for the implementation of the basic method. Task graphs are scheduled, the schedules are turned into budgets and the <em>n</em> best are kept.</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>The same task is present in two independent cases with possibly different constraints.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>The two mirrored tasks C are removed at the same time and replaced with deadlines and release times.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Tasks with different periods could be rolled out to same period length. In the representation to the left, the period for task <em>T</em>2 is half of the period of task <em>T</em>1. In the representation to the right, all tasks have the same periodicity and task <em>T</em>2 has been rolled-out into two instances.</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>A task set with shorter period than deadline. If all tasks are allocated on same processor pipelining cannot be used (i.e. not optimal). In other words, the allocation in the left part of the figure might be impossible to implement.</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>A task set with shorter period than deadline. Pipelining is possible without altering the allocation of the different task instances.</td>
<td>27</td>
</tr>
</tbody>
</table>
4.1 The theoretical tightness limit, \( t_{\text{lim}} \), for the different cases. The values are normalized to the minimum tightness, \( t_{\text{min}} \), in the long experiments (i.e. the estimated optimal tightness), for each case. 34

4.2 The results of the ten-hour experiments using all the different search methods. The values are normalized to the minimum tightness, \( t_{\text{min}} \), found in the three long experiments (i.e. the estimated optimal tightness), for each case. The linear and multi-linear search method did not find a feasible solution for the reconfigurable case. 36

4.3 The results of the one-hour experiments, using all the different search methods. The values are normalized to the minimum tightness, \( t_{\text{min}} \), in the long experiments (i.e. the estimated optimal tightness), for each case. The linear and multi-linear search method did not find a feasible solution for the reconfigurable case. 36

4.4 The results of the experiments using the random search method for different durations. The values are normalized to the minimum tightness, \( t_{\text{min}} \), in the long experiments (i.e. the estimated optimal tightness), for each case. 37

4.5 The diagram shows how many iterations are needed to make the budgets reach within 5\% of \( t_{\text{min}} \). 37

4.6 The deviation from \( t_{\text{min}} \) for short random searches does not show a clear trend, as we had expected. This is suggested to be examined further in future work by adding a number of cases of greater size. 38

4.7 The model for the implementation simulation consists of three parts, budgeting method (Bx), implementation method (I) and schedulability analysis (SA). The input TD is data from the task dimensioning. If implementation fails and no alterations can be made, FAILURE is signaled. When implementation is finished, a schedulability analysis is applied. If the system is schedulable, SUCCESS is reported. 39

4.8 The relative cost of using the budget-based method drops for systems of larger size. 43

4.9 For failing implementations, the trend that the relative cost drops for systems of larger size is even more distinct. 44
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The properties in a typical task graph. Each task, ( M ), has an estimated execution time, ( v_e ). The end-to-end constraints are noted as offsets, ( v_o ), and deadlines, ( v_d ).</td>
</tr>
<tr>
<td>2.2</td>
<td>The tightest path has been found. The end-to-end constraints are derived into AETs, ( v_{AET} ), for each task on the path.</td>
</tr>
<tr>
<td>2.3</td>
<td>The second tightest path has been found. The end-to-end constraints are derived into AETs.</td>
</tr>
<tr>
<td>2.4</td>
<td>All tasks have been assigned AET’s. Note the communication task generated (labelled C).</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of the results of the experiments, showing the relative sizes of the test cases and the tightness of the best budgets found. &quot;lin&quot; stands for linear search, &quot;mlin&quot; stands for multi linear search, and &quot;rnd&quot; stands for random search. The tests are divided into three categories based on their execution time. Value ( t_{lim} ) indicates the absolute limit for the tightness, and ( t_{min} ) indicates the minimal achieved tightness. Value (</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Wouldn’t it be great if there were a systematic method for deriving the non functional constraints available at design time that made it possible to verify design and make implementation a much clearer task?

Developing a computer system is a very complex task, where there typically exist several constraints, interacting in a seemingly non predictive manner. This alone makes the process of computer system design very hard, even if only fully implemented software components were used. In the typical case, a few components may already be implemented, evaluated and documented (i.e. bought or reused), but most of the components will initially be bare specifications.

For modern embedded real-time systems, complexity depends on functional requirements (e.g. the brake functionality of a car) and non functional requirements (e.g. the maximum allowed jitter between the different brake circuits) [6] [3]. Since traditional system development methods handle only functional constraints and as the complexity of non functional constraints steadily increases, other types of development methods are needed. Related research shows that the lack of validation of non functional constraints, as in the case of traditional development methods, will be very expensive in the long run [8]. In the worst case the project ends with a system implementation that fails to fulfill the non functional constraints of the system at the point of execution because of a faulty design. This in
turn would result in the project most probably being abandoned or re-engineered using a new design. Either way, it is more expensive to detect errors after implementation than it is to detect bad designs directly, since a decision can be made at the source of the problem – whether to abort the project or start over. Note that iterative development is probably unavoidable, but the iterations should be as short and cheap, in terms of work hours, as possible.

Besides the complexity issue, the design of tasks and the dimensioning of the resources are done at a point in the project when resource limitations may be unknown and tasks have not yet been implemented [13]. A method is thus needed to guide the designer, to keep within the specified non functional constraints, and validate the design, guaranteeing that there will be a way to implement and execute the system [2]. Since many parameters are unknown until implementation is completed, we would like to keep the design flexible for as long as possible to avoid dead ends in the development [10] [9] [16].

This thesis models project development as consisting of the following four steps:

1. Requirement specification. Specifying the requirements as given from the system specification. Determining end-to-end constraints.
3. Implementation. Implementing all tasks.
4. Execution/use. Integrating tasks into a system and deploying the system.

To avoid dead ends and project breakdown, work in each step of the development must be validated before allowed to continue (see Figure [1.1]). For example, there is no point in implementing a system that is impossible to schedule and execute within the specified requirements. The validation should preferably not only be used to decide whether the developed system is fulfilling the initial requirements, but also to decide how much the current solution could be adjusted, and still fulfill the requirements. To be able to compare solutions, a measure of the flexibility is introduced. This makes it possible to decide whether a design that only results in solutions of low flexibility may have to be re-engineered to avoid future dead ends. Flexibility must be considered for functional constraints and for non functional constraints [18]. However, well defined frameworks and methods exist for functional constraints, and this thesis will therefore focus on the handling and derivation of the non functional constraints.
1.1. BACKGROUND

![Figure 1.1: Overview of the suggested development flow model with requirement forwarding. Validation of the design at an early stage makes it possible to avoid future dead ends.](image)

Figure 1.1: Overview of the suggested development flow model with requirement forwarding. Validation of the design at an early stage makes it possible to avoid future dead ends.

![Figure 1.2: In ad hoc development methods the non functional constraints will not be used to verify the system until after implementation.](image)

Figure 1.2: In ad hoc development methods the non functional constraints will not be used to verify the system until after implementation.

For example, a typical unstructured approach to system development is the ad hoc approach, where a system must be fully implemented before any validation can be done. In the development model this is represented by forwarding the non functional constraints directly from the requirement specification to the execution validation stage (see Figure 1.2). This means that faults will be detected, at earliest, after implementation. If a failure is detected at this late stage, modifications will include, at least, debugging and re-implementation [19]. If the underlying problem is a bad design, costly re-design will be needed.

Another type of development method is where the non functional constraints are used to validate the implementation. This has been an active research area in the past 15 years and many useful methods have been developed [4] [1], of which a few are actually accepted and used in industry [11]. This area of research is
CHAPTER 1. INTRODUCTION

Figure 1.3: Ad hoc development makes use of the non functional requirements at the implementation stage (for validation).

usually known as scheduling and schedulability testing. The validation methods often need information such as worst case execution time (or at least an estimate of the worst case execution time) and perhaps even processor allocation. The scheduling approach makes it possible to detect errors in design earlier than in the ad hoc method (see Figure 1.3), but it still might be necessary to loop back to dimensioning to solve underlying problems.

Research in the area of design validation has led to ways of detecting the problems at the source. If a design has been found to be impossible to implement with the non functional constraints, it is then only the design that has to be re-engineered (see Figure 1.4).

Summaries of further related work are found in Appendix A.

1.2 Focus and limitations

This thesis focuses on how to solve the problem of derivation, handling and validation of non functional constraints during design and implementation, although many system parameters are still undefined. A method for conducting constraint derivation and validation has been developed and implemented, and it is meant to be used between design and implementation. The non functional requirements, in combination with a coarse functional description (a task graph), have been used
to generate detailed (task level) implementation constraints. The properties of the implementation constraints that are generated are the following:

- The implementation constraints are based on relative and rough estimates of the implicit constraints.

- If it is possible to implement all tasks within their derived constraints, there exists at least one feasible execution schedule.

- The implementation constraints are derived in such a way that the probability of succeeding with system implementation is sought to be maximized.

What makes this method different from using a standard allocating optimal scheduler with execution time estimates as input is that: (i) a scheduler needs absolute timing estimates, and not relative ones, as in our proposed method; (ii) the scheduler cannot generate implementation constraints unless there exists a schedule within the resource boundaries. The method presented in this thesis is always able to generate implementation constraints for a task graph. However, there is no guarantee that it is possible to fulfill all constraints during implementation.
1.3 Contributions

If you do not know where you are going, does it matter which way you go? This is more or less the very problem that a developer of any computer system will experience. At the initial stages of the development, there is very little information about the actual implementations of the future functions, which means that it is difficult to derive the components’ constraints. This in turn makes it difficult to make decisions about the hardware, or the resources needed for the tasks to be implemented in the system being constructed.

Late detection of problems in the design comes at a high cost. The urge to avoid these problems has resulted in the development of methods and tools that help system designers and developers in different ways to detect problems as early as possible during system development. To avoid expensive surprises late in a project, design should be focused on finding solutions maximizing the probability that it will be possible to implement the system. In this thesis, some form of solution space exploration is used to find the most flexible resource allocations in the design phase. As the developer becomes more certain about the components of the system, more becomes known about the system as a whole, and the resource estimates become resource requirements. It will thus be possible to understand where the project is heading and, most important of all, whether it is possible to complete the project at all.

The contribution of this thesis is an extension to the field of embedded computer system development by a method that makes it possible to detect system design and dimensioning problems even before the system’s components are fully implemented. The method extends the tightness metric and method proposed by Y. Zhao [23] with, for example, allocation support and estimates of different certainty. A lot of work has been put in the development of implementations of the proposed method, extensions to the method, and simulators for evaluation of the implemented method. Evaluations are made of the method’s practical usefulness, scalability and usability. To streamline and simplify the evaluation, this thesis focuses in particular on real-time latency requirements and simple homogeneous single bus platforms. However, the principles presented can easily be extended to wider classes of constraints and systems. The results from this research have been presented at several national and international conferences, seminars and workshops (e.g [20, 22]). The following is a summary of the main contributions.

- Presentation of a concept for implementation constraint derivation (ICD).
1.4. THESIS OVERVIEW

- Implementation of a basic ICD (BICD) method, based on the work of Y. Zhao [23].
- Demonstration of the practicality of BICD in terms of scalability.
- Demonstration of the possibility to find near optimal solutions in a reasonable time.
- Demonstration of the usefulness of the BICD method in terms of implementation cost reductions.
- Development of a method used for simulation of the software implementation process.
- Presentation of several extensions of the BICD that make it possible to handle systems in a more realistic fashion.

1.4 Thesis overview

The thesis is structured as follows. Chapter 2 describes the details of the problem of deriving component constraints. A detailed description is given of the BICD method and how component constraints are generated. The chapter ends with an example of how to apply and use the BICD, along with a discussion about the complexity of different systems. Chapter 3 describes how the BICD method can be extended to be able to handle more realistic systems. It is shown how to handle multiple task instances, how to handle periodic systems, how to state variation in parameter quality and how to support various types of non functional constraints. Chapter 4 discusses the aspects of the BICD method that must be evaluated. Methods for how to make the actual evaluations are presented. The chapter concludes with the results of the evaluations along with a discussion of the experiments conducted. The thesis is concluded in Chapter 5 with discussions of the benefits of the BICD method and a short discussion of the evaluations. The chapter also presents some suggestions about how to continue this research.
Chapter 2

The Basic Implementation Constraint Derivation (BICD) Method

2.1 Introduction

As described in Chapter 1, an increasingly important problem in the development of embedded computer systems is that bad system design will result in dead ends. In the worst case, this problem remains undetected until final testing, after implementation and integration of all components (or even later - at execution). This is of particular interest when it comes to meeting non functional constraints, such as performance or resource utilization requirements, since even though a system may be functionally correct it is useless if the non functional constraints cannot be fulfilled. To correct a problem at a late stage of the development is very costly, as it often requires re-engineering both of the design and implementations. This situation becomes more critical as the system complexity (i.e. the number of interactions and dependencies between different system components and system requirements) increases.

The common (traditional) method for handling non functional constraints in embedded computer system design is *ad hoc*. Very little systematic or formal analysis of real-time properties (such as worst case response time or the ability to meet deadlines) is made, for example. Such properties are instead checked after im-
CHAPTER 2. THE BICD METHOD

plementation in testing. In recent years, however, new analysis techniques have been developed that allow several properties to be predicted in a more systematic and formal way. Especially in the case of real-time systems, these analyses often rely on the assumption that some runtime constraints are enforced, such as individual task deadlines. A well-known example of this kind of real-time analysis technique is Rate Monotonic Analysis (RMA), first described in [11]. With these analysis techniques, systematic methods to derive the required runtime constraints have also been proposed, see for example [12, 15]. These methods, of predicting whether a system will meet its constraints, make it possible to detect run-time problems in the implementation phase. The methods may also be useful for pointing out a system’s design or implementation related problems. However, these methods typically rely on an existing implementation of a system’s software components, requiring for example that the worst case execution time (WCET) for each component is known. This means that fundamental design problems may remain undetected until all components have been implemented.

To avoid the drawbacks of ad hoc and schedulability analysis, it is possible to use a method able to derive guidelines at design, to be used at the implementation, such that keeping within those guidelines will maximize the probability that the overall system requirements can be met. In that way, failing to keep within the guidelines will be an immediate warning signal, meaning that component development should be postponed until any necessary modifications to the design or guideline assignment has been made. Guidelines could also be used to decide the way in which components are implemented or compiled, by suggesting trade-offs between, for example, execution time and memory consumption. Later in this thesis, the guidelines will be referred to as implementation time constraints (ITCs). Note that this should not be confused with the run-time constraints (RTCs) derived later, at scheduling.

This thesis describes a method for ITC computation along with the results of evaluations of the method. The evaluations are based on a prototype implementation of the ITC computation method in the form of a software tool implemented in Java.

2.2 Problem statement and assumptions

As written above, the problem is to find a method that, based on a functional system description with end-to-end constraints and a given platform, generates a set
of task level ITCs (which will optimally maximize the probability of successes of the implementation). The functional system description is in the form of a directed acyclic task graph, where the designer has estimated the execution time for each task in the graph in relative terms. Each task can have optional timing constraints (i.e. in the form of release time and/or deadline) and optional locality constraints that limit the possible processor allocations. For each precedence constraint of a task, an optional communication package of estimated relative length can be added. An example of the way this would look is given in Table 2.1.

The platform is modelled as a network with a number of processors connected to it. In this thesis, we assume homogeneous processors connected to a single bus, meaning that the only platform parameter is the maximum allowed number of processors. The motivation for this restriction is that, although the model is somewhat simplified, it will still be possible to validate the usefulness of the method.

The output of the method is a set of budgets where each budget in the set consists of one ITC generated for each task and for each associated communication tasks in the task graph. In this thesis, resource utilization constraints are limited to include execution time and processor allocation constraints. It is also assumed that the relative deadline, $D$, for a task does not exceed the task’s period, $P$, i.e. $D \leq P$.

### 2.3 The BICD method

To be able to find optimal or near optimal budgets it is necessary that the search method has the potential of being able to search through all possible processor allocations and execution orders (schedules) for tasks according to the constraints.
The need to be able to check all possible schedules stems from (i) that execution order and allocation are the parameters that affect the optimization metric and (ii) that we would like to know whether we, after implementation, will be able to generate a schedule for the system that enables us to make guarantees about whether it will be possible to fulfill the end-to-end constraints. An overview of the method is shown in Figure 2.1 and further described below.

2.3.1 Method

The input data (including resource parameters, performance restrictions, locality constraints and task set) are preprocessed, generating a constrained task graph. The preprocessor is followed by the generation of schedules based on the constrained task graph (scheduler in the figure) and consists of three steps: ordering, allocation and generation of communication. The purpose of the ordering is to generate permitted execution orders for the tasks which fulfill the precedence constraints of the task graph. The purpose of the allocation is to allocate each task to an explicit processor according to the locality constraints and within the maximum allowed number of processors. The purpose of the communication generation is
2.3. THE BICD METHOD

to model the data transactions that take place on the shared medium. Communication tasks are generated on the basis of the allocation information, since this indicates which tasks have to communicate using the bus. Finally, ITCs (budgets) are generated for all tasks on the basis of the scheduled task graphs. The top \( n \) budgets according to the optimization metric (explained below), where \( n \) is the number of budgets that we would like to keep, are saved in the set of budgets that are the out-data from the budget generation method. As the possible schedules are quite many, and as this scales quickly with the number of tasks in the task graph, a plain search is not good enough. Several different heuristics that makes it possible to fins near optimal solutions by evaluation of just a few of all possible schedules have thus been proposed and evaluated (see Section 2.4).

The optimization metric is equal to the graph tightness, \( T \), which is a function of the scheduled task graph, \( G \). The scheduled task graph \( G \) consists of tasks, \( V \), and precedence constraints, \( E \), for which there exists a number of different execution paths, \( p \in P \). The precedence constraints \( E \) consists of the constraints from the task set and also the constraints that have become explicit through scheduling. The graph tightness is equal to the maximum tightness for all paths, where the tightness for a path, \( p_T \), is equal to the fraction of total work, \( W \), relative to the path length, \( L \). The total work on a path is equal to the sum of the estimated worst case execution time (EWCET), \( v_e \), for all tasks on the path and the path length is equal to the difference between the deadline and offset for the path, \( p_d - p_o \). The path’s offset is equal to the first task’s offset, and the path’s deadline is equal to the last task’s deadline. In summary, we have the following definitions:

\[
G = < V, E > \tag{2.1}
\]
\[
W = \sum_{v \in p} v_e \tag{2.2}
\]
\[
L = p_d - p_o \tag{2.3}
\]
\[
p_T = \frac{W}{L} \tag{2.4}
\]
\[
T = \max(p_T : p \in P) \tag{2.5}
\]

When the path with the maximum tightness in the task graph has been found, the ITCs for the tasks on this path could be generated. To generate the ITC for each task on the path, the estimated execution time, \( v_e \), is divided by the path tightness \( p_T \). This means that the sum of all ITCs for the tasks on this tightest path equals the length, \( L \), of the path. The ITC generated is called the allowed execution time (AET):

\[
v_{AET} = \frac{v_e}{p_T} \tag{2.6}
\]
Explicit offsets and deadlines are generated for all tasks on the path since all tasks have AETs and the path has an explicit offset and deadline, respectively. The method chosen to calculate the offset and the deadline explicitly for a task on the tightest path is to set the offset for a task to the previous task’s deadline and the deadline to its own offset plus its AET:

\[ v_{io} = v_{(i-1)d} \]  \hspace{1cm} (2.7)

\[ v_{id} = v_{io} + v_{iAET} \]  \hspace{1cm} (2.8)

When all tasks on the tightest path have explicit offsets and deadlines they are removed from the task graph and replaced by the corresponding deadlines and offsets. This leaves a task graph where resources are reserved for the tasks removed. The tightest path in this new reduced task graph is calculated and the corresponding AETs are generated. This procedure is then repeated (i.e. finding the tightest path, calculating AETs and replacing budgeted tasks) until all tasks have been assigned AETs. Note that the tightness of the whole task graph is defined as the first tightest path’s tightness. The consecutive paths’ tightnesses are not needed to rank the task graphs but are calculated for the generation of all tasks’ AETs (which may be postponed for calculation until needed).

Although the maximum probability for implementing according to the constraints is searched using the task graph tightness as the optimization metric, it can still be difficult to implement according to the budget. The guarantee is that, if all implementations of tasks in a task graph fit within the corresponding ITCs, there exists at least one schedule for the system. If an implementation cannot be made within the constraints of a budget, there is still a possibility that there exists other budgets with the same or higher tightness that it is possible to keep within. To reduce the need to start the whole budgeting process over when implementation fails to comply with a budget, a whole set of promising budgets is generated and kept for later use.

2.3.2 Example

To illustrate how budgets are generated for a task graph, we calculate the ITCs step by step for the task graph in Table 2.1. In this example, tasks $M_{n-0}, M_{n-2}$ and $M_{n-4}$ are allocated to processor 1, while tasks $M_{n-1}, M_{n-3}$, and $M_{n-5}$ are allocated to processor 2. The execution order in each processor is (in this case) given explicitly by the precedence constraints in the task graph. However, since
2.3. THE BICD METHOD

Table 2.2: The tightest path has been found. The end-to-end constraints are derived into AETs, $v_{AET}$, for each task on the path.

<table>
<thead>
<tr>
<th>$M_n$</th>
<th>$v_o$</th>
<th>$v_d$</th>
<th>$v_e$</th>
<th>$v_{AET}$</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-1</td>
<td>10</td>
<td>52</td>
<td>40</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>n-3</td>
<td>52</td>
<td>73</td>
<td>20</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>n-5</td>
<td>73</td>
<td>100</td>
<td>25</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>

the two communicating tasks $M_{n-0}$ and $M_{n-3}$ are allocated to different processors we have to create a communication task, $C_{n-0,n-3}$, allocated to the communication processor (the bus seen as a processor). It is assumed that, according to the specification, the new task, $C_{n-0,n-3}$, has the estimated communication time of 5. To calculate the tightness and assign AETs for all tasks, we have to find the tightest path in the graph. There are three different paths in this example, $p_{n-0,n-4}$ with the total work of 125 and length of 150, $p_{n-0,n-5}$ with the total work of 75 (including the communication task $C_{n-0,n-3}$) and length of 100, and $p_{n-1,n-5}$ with the total work of 85 and length of 90. The tightness is calculated for each path (0.667, 0.75, and 0.944, respectively), showing that $P_{n-1,n-5}$ is the tightest path with a tightness of 0.944.

Since the calculations show that path $\{M_{n-1}, M_{n-3}, M_{n-5}\}$ is the tightest it will also be the first to be assigned AETs (see Table 2.2). The assigned tasks are then removed from the task graph and replaced with release times and deadlines where these apply. Here it means that the communication task, $C_{n-0,n-3}$, will get a deadline of $100 - 27 - 21 = 52$. The tightness calculations are now repeated for the remaining tasks in the graph. Path $\{M_{n-0}, M_{n-2}, M_{n-4}\}$ is now the tightest and is therefore assigned AETs (see Table 2.3). The tasks are then removed from the task graph and replaced with a release time for $C_{n-0,n-3}$ that will start at $0 + 30 = 30$. Now there is only one task left ($C_{n-0,n-3}$) that has a release time and a deadline, which makes the generation of its AET trivial ($52 - 30 = 22$). All the tasks in the graph have now been assigned AETs (see Table 2.4) and the tightness for this graph is equal to the tightness for the tightest path (0.944).
CHAPTER 2. THE BICD METHOD

Table 2.3: The second tightest path has been found. The end-to-end constraints are derived into AETs.

<table>
<thead>
<tr>
<th>$M_v$</th>
<th>$v_o$</th>
<th>$v_d$</th>
<th>$v_c$</th>
<th>$v_{AET}$</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-0</td>
<td>0</td>
<td>30</td>
<td>25</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>n-2</td>
<td>30</td>
<td>90</td>
<td>50</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>n-4</td>
<td>90</td>
<td>150</td>
<td>50</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: All tasks have been assigned AET's. Note the communication task generated (labelled C).

<table>
<thead>
<tr>
<th>$M_v$</th>
<th>$v_o$</th>
<th>$v_d$</th>
<th>$v_c$</th>
<th>$v_{AET}$</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>n-0</td>
<td>0</td>
<td>30</td>
<td>25</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>30</td>
<td>52</td>
<td>5</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td>10</td>
<td>52</td>
<td>40</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>n-2</td>
<td>30</td>
<td>90</td>
<td>50</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>n-3</td>
<td>52</td>
<td>73</td>
<td>20</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>n-4</td>
<td>90</td>
<td>150</td>
<td>50</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>n-5</td>
<td>73</td>
<td>100</td>
<td>25</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>
2.4 Complexity

Since the search problem scales very quickly with the size of the task graph, both in terms of possible schedules and in terms of possible paths, the search efficiency needs to be improved using some kind of heuristic (the search problem is similar to multiprocessor allocation, which is an NP-hard problem). The search space is limited by all possible orders of the tasks, where there exists several possible allocations for each order and searching through all possible schedules (linear search) for a task graph quickly becomes too slow. Several alternative search methods are therefore evaluated and described below.

Linear search

Linear search describes the search mode where the search is performed in iterations from a starting point, one step at a time, changing the schedule a little each time in a deterministic way. This search will always result in the same search from time to time. The advantage of the linear search is that, without run-time limitations, the whole search space has eventually been evaluated. Ordering in the linear search is done by picking the tasks in order from left to right in the task graph, breadth first. Allocation is made depth first, generating all possible allocations of the last task in the generated order before altering the allocation of the next to last task.

Multi-linear search

Multi-linear search is a set of linear searches started at random points in the search space. After a period of time, the number of traces will be pruned by removing the worst cases and the rest of the traces will then continue until the next pruning and repeat until a desired number of traces are available. Since the start points are random, this search generates different solutions each time. Besides random starting points, the multi-linear search behaves like the linear search but possibly with faster progress, since several linear searches are made in parallel.
CHAPTER 2. THE BICD METHOD

Random search

Random search is simply a method where the schedule is chosen at random for every budget. The drawback is that the random method cannot make any use of the knowledge of good solutions, it is completely random.

Branch and bound

Branch and bound (B&B) may seem to be an efficient and well tested method for generating feasible schedules quickly. For budgeting, however, this will not be of any real help. The problem is the tightness optimization of the ICD method, where the tightness is based on a complete schedule. This makes it difficult to back-propagate information about the tightness to the scheduler, as is needed for B&B.

Neighborhood search

Neighborhood search is used to make a local optimization, in the neighborhood of a good solution. To achieve this, a method is needed for making small modifications to the schedule, which could be implemented as an adaption of the linear search method. However, since the modifications are very small, it may be possible to predict whether the tightness will increase or decrease for a certain schedule modification. This would speed up the search significantly, although it will not be optimal.

Unguided algorithms

Genetic algorithms, simulated annealing and other unguided methods will most probably show poor performance in the evaluation of schedules with respect to tightness. The reason for this is that the model used in the calculation of the tightness is hard to convert to a model that can be used by the unguided algorithms. However, these algorithms may be very efficient for the allocator part of the scheduler.
2.5. USE OF BICD

The conclusion of this summary of search methods is that different heuristics should be used in different phases of the method. For example, a specific type of allocation could initially be evaluated (feasible allocations with maximum slack) using B&B, hopefully finding good solutions in a shorter time. These initial solutions could then be used as seeds for a neighborhood search.

2.5 Use of BICD

The budgets generated by the BICD method are meant to be used as guidelines when the tasks are to be implemented. Although near optimal budgets are produced from the BICD method, there is no guarantee that all task implementations will fit within their budgets. Two methods for handling this are proposed: budget combining and budget trade-offs.

As mentioned above, the proposed basic budgeting method is meant to generate budgets with an optimal tightness. However, there may exist implementations that are feasible but not optimal according to the tightness metric. To add stability against future alterations of the implementations and to give the developer a hint about where the design will be critical (i.e. where to start implementing), multiple budgets (of any tightness) are saved as a set of budgets. As implementation is carried out, budgets are removed from the set as they are exceeded, leaving only the feasible budgets. For systems based on good estimates, the implementation will most probable be completed before all budgets are exhausted. Otherwise, another set of budgets is generated, possibly with updated information about the execution time estimates.

One suggestion is for multiple budgets to be combined into one budget overview for the developer, reflecting all the budgets’ characteristics. Allowed execution time could be reflected by minimum and maximum allowed execution times for each task, for all budgets. Allocation could be reflected as the set of processors representing all currently budgeted allocations for the task.

If some tasks are implemented within their budget by a margin, and some tasks are implemented exceeding their budget, it is possible to trade off surplus and shortage. If the trade-offs are successful, an expensive iteration has been avoided. Two criteria should be fulfilled in making trade-offs in the budgets generated.

- End-to-end constraints will have to remain fulfilled after trade-offs.
All tasks’ tightnesses are limited by a pre-defined trade-off tightness limit.

The first criterion is self explanatory. The second criterion means that, although it may be possible to trade off surplus, it may still be prohibited with respect to the maximum allowed overall tightness set by the developer.

To keep within end-to-end constraints after the trade-off, the method is split into two cases.

- When trade-off is done for two tasks where the source of the trade-off (assumed to be the parent) only connects to a single child, the destination of the trade-off.
- When trade-off is done for two tasks where the source (assumed to be the parent) of the trade-off has more children than the designated destination of the trade-off.

In the first case the trade-off is done freely between the two tasks without a need for any further action. In the second case the trade-off has to be distributed to all tasks to which the source is connected. Since the tightness of the task graph is changed when trade-offs are made between tasks, a maximum allowed tightness is set for the task graph before trade-offs are done. This makes it possible to calculate the minimum and maximum allowed AET for each task.

Although the two proposed methods, budget combining and budget tradeoffs, increase the usefulness of the budgets generated by the BICD method, these methods still do not find better budgets. There are several extensions of the model and the BICD method that make the budgets more realistic (e.g. handling periodic systems and estimates of different precision). This will be discussed in Chapter 3.
Chapter 3

Extending the BICD

3.1 Introduction

The BICD method discussed in the previous chapter works under the assumptions that each task is represented once in the task graph, all tasks share the same periodicity, all estimates are of the same precision and the only resource-derived constraint is time. In real projects, however, there is a need to describe a system as several different independent task graphs, each addressing one aspect of the system and one set of constraints at a time. For example, in Figure 3.1, the task \( B \) is used in two different contexts. Note that, in the example shown in the figure, the allowed execution paths will be either \( \{ A, B, C \} \) or \( \{ B, D \} \). Further, tasks of different periodicities should be allowed as separate groups in the same task graph. Since reuse of components is common, a reasonable assumption would be that the estimated execution times will not be of the same certainty for all tasks. Extensions that handle these shortcomings are discussed in the following sections.

3.2 Multiple task instances

For a real design it is likely that the specification consists of several different sets of independent and dependent constraints which should all be fulfilled. It is very difficult (or even impossible) to model this as a single set of constraints for a task graph, where each task has only a single instance. One reason is that it may be
impossible to reduce several sets of constraints without taking the schedule in account. A typical design would rather be based on several different use cases, where each case is described as a task graph, or several task graphs, with several constraints applied to that specific task graph. To be able to describe these multiple independent sets of constraints, a slight modification is made to the task graph, introducing the multiple task instance representation (MTIR). The MTIR allows sets of task graphs to be budgeted and allows the same task to appear several times, which makes it possible to describe constraints that vary for tasks in different situations.

Two different instances of the same task will have the same properties (i.e. allocation and esitmate) since both instances will execute the same code. This will be a problem when one task instance ends up on the tightest path and should be assigned an AET. As one instance of the task is assigned an AET, all other instances should be assigned this tightest AET at the same time. As all instances of the task have been assigned an AET they should also be removed. However, to remove a task, we need to know the offset and deadline, which are only guaranteed to exist for the tightest path.

To solve the problem described, the following observations are made.

- For any given path on the system, the length of this path, \( L \), will correspond to the sum of all tasks’ allowed execution times, \( AETs \), after budgeting.
- We are free to assign a lower \( AET \) to a task if we would like to.
3.2. **MULTIPLE TASK INSTANCES**

- We are free to remove a task at any time if correct offset and deadline could be assigned.

On the basis of these observations, a method for removal of tasks with multiple instances (mirrored tasks) has been formulated. The problem is that as soon as a task has been assigned an *AET* it is reflected to all mirrored tasks. This may lead to several budgeted tasks in the task graph that we would like to remove, although they have no explicit offset or deadline.

The problem can be divided into two parts, the case where all mirrored tasks have a deadline or an offset, and the case where the remaining budgeted task lacks at least either offset or deadline. In the first (trivial) case, where we have some kind of reference coupled to the task, it is possible simply to remove the task and replace it with an offset and a deadline.

In the second case, where no explicit offsets or deadlines are known, we have to calculate the worst case offset and deadline on the path of this task. To do so we calculate the sub-tightness, $t'$, for each possible path through the task we would like to remove, $m$, where the sub-tightness is the tightness for a path where the mirrored task has been omitted:

$$t' = \sum_{i \neq m} c_i \quad \frac{L - AET_m}{(3.1)}$$

On the basis of the tightness of the sub-tightest path, we now divide the remaining time on the path over the tasks. This can be done since the total length of a path represents the sum of all *AET*’s and an *AET* is already assigned to one task (task $m$) on the path.

After that, the offset and deadline for the task we are about to remove are set. It is important to point out that the *AET*’s for the other tasks (i.e. on the path) are not affected at this point; it is only the task to be removed whose offset and deadline are updated.

If there is more than one mirrored task instance that should be removed, a minor adjustment to the method is needed. Sub-tightness $t'$ then has to be calculated for all mirrored task instances. Following this, the tasks can be assigned offsets and deadlines and be removed in descending order of sub-tightness.
Figure 3.2: The two mirrored tasks C are removed at the same time and replaced with deadlines and release times.

Example

If the system being designed is described as the graphs in Figure 3.2 where a task C is mirrored into two instances (C_a and C_b), there could be a problem when one of the task instances of C is about to be removed from the tightest path. In the example, the tightest path involves task instance C_b, and we have to remove instance C_a by applying the described method extension.

First, the tightest path is found (B, C_b, E) and budgets (b', c', d') are calculated for each task on the path. All paths that connect through mirrors of "the task to be removed" (i.e. C_a) will be evaluated for their sub-tightness, t'. The sub-tightest path is isolated (A, C_a, D) and the release time and deadline for the task instance C_a are calculated (r_2, d_2). Task instance C_a can now be removed from the task graph and replaced with release time and deadline, as seen in the right half of Figure 3.2.
Figure 3.3: Tasks with different periods could be rolled out to same period length. In the representation to the left, the period for task $T_2$ is half of the period of task $T_1$. In the representation to the right, all tasks have the same periodicity and task $T_2$ has been rolled-out into two instances.

### 3.3 Periodic systems

The problem of handling periodic tasks as part of the system, where the period differs from task to task, is divided into two parts. First, the tasks should be adjusted in such way that all tasks receive the same periodicity through roll-out, as described below. Second, since roll-out results in multiple instances of the tasks, there will be problems if all task instances are not part of the tightest path (removed at once).

The first part of the problem is solved by applying roll-out. Roll-out is done by adding instances of all tasks until the least common product (LCP) of all tasks’ periods is reached (i.e. all tasks get the same period, but a different number of replicated instances). Since a schedule for a rolled-out task graph will be repetitive after the first LCP period, only this portion has to be analyzed (see Figure 3.3). Note that the length of this rolled-out cycle will be equal, or less than, the product of all periods in the system. Since the length of the roll-out is determined by the mutual period multiplicity, we have to find a period length for which all periods are evenly divideable. This can unfortunately result in very long roll-outs for task graphs with periods of prime multiplicity.

The second part of the problem is to remove all mirrored task instances when one task instance is removed. Offsets and deadlines for instances that are not on the tightest path are simply calculated using the same method as used for removal of
task instances in MTIR.

There will be several situations where roll-out according to the described method will not be resource efficient. The case where tasks that have a deadline after the invoking period, roll-out according to the method described above, will result in all instances of that task being allocated to the same processor. This is because different task instances are supposed to have the same properties. It may be more efficient to allow pipelining of the tasks by allowing different instances of the same task to be allocated on to different processors, see Figure 3.4. To allow pipelining of the tasks a modification of the task model is needed where mirrored tasks do not have to be allocated to the same processor. A problem that could occur when allowing pipelining is that some tasks will have deadlines after the rolled-out period ends. Since the schedule is supposed to wrap around at the rolled-out period length, a deadline that extends over this point is only allowed in the case that corresponding free space is present at the start of the period (explained below). Note that this way of handling pipelining also could be used for the case where tasks are allocated on different processors from start (see Figure 3.5).

To be able to calculate the free space and how much a task exceeds the rolled-out period length, release time for the first task and deadline for the last task on each processor are needed. If either release time or deadline is unknown on a processor it is possible to calculate the missing parameter. This is done by updating the deadline for the last task to the release time of the first task on the processor plus the length of the rolled-out period, for each processor. If a release time is missing, it would be possible to do the reverse: calculate the offset of the first task on the processor as the last tasks deadline minus the length of the rolled out period.

However, there will be a problem if neither the first nor the last task on the processor has an release time or a deadline. The method for solving this is to insert an release time equal to zero and a deadline equal to the rolled-out period for the tasks allocated on the processor. This approach is not optimal but will at least make sure that it will be possible to fit all tasks allocated to a processor without explicit release times and deadlines.

Finally, when budgeting, mirrored rolled-out task images will be dealt with in the same way as in the case of multi case task graphs – simply by updating the allowed execution time for all mirrored tasks at the same time.
Figure 3.4: A task set with shorter period than deadline. If all tasks are allocated on same processor pipelining cannot be used (i.e. not optimal). In other words, the allocation in the left part of the figure might be impossible to implement.

Figure 3.5: A task set with shorter period than deadline. Pipelining is possible without altering the allocation of the different task instances.
3.4 Varying quality of the estimates

Using the basic method, the distribution of "slack" is done proportionally to the tasks' estimate. The motivation for this is that a task with a large estimate is assumed to be proportionally more wrong than a task with a small estimate. This may not be true at all when thoroughly specified and compiled software is purchased or old components are reused. Since the values used as EWCETs for known tasks will be very close to the WCET, extra time is not needed. However, it is crucial that the AET is at least equal to the EWCET. In cases such as this, where there is greater knowledge about the estimates, a correctness parameter is introduced to reflect the precision of the estimate.

The basic form of the AET equation is transformed to the form \( \text{estimate} + \text{portion of slack} \):

\[
T_{AET} = e_n + \left( L - W \right) \times \frac{\text{portion}}{W}
\]

(3.2)

(3.3)

To adapt this equation for a task graph with estimates of varying correctness, the estimate is simply swapped for correctness, \( \delta \), and the total work on the path for the sum of all correctness variables on the path, \( \Delta \):

\[
T_{AET} = e_n + \left( L - W \right) \times \frac{\delta}{\Delta}
\]

(3.4)

In other words, instead of adding the most time to the task with the largest estimate, the task for which the least is known (i.e. which has the maximum uncertainty) will receive the maximum contribution.

3.5 Other types of requirements

The BICD method handles only the assignment of time budgets, but parameters such as memory and energy should be of equal interest, especially in embedded computer systems.
Budgeting memory for tasks may be done in a similar way as in time budgeting. Each task will have to have an estimated memory consumption, and the end-to-end timing constraints are replaced by the corresponding, per processor memory size. This can be modelled using an offset of zero and a deadline equal to the memory size. Since the allocation of tasks on the different processors will not be known until after the ICD method has started, the addition of memory constraints would have to be a step similar to the communication addition step, where communication tasks are added (the memory constraints are specified for the processor).

When it comes to energy, it may not be interesting to assign energy constraints or energy budgets but rather to calculate an estimate of the energy consumption for a certain set of ITCs. It would be possible to calculate a rough estimate of the energy consumption using the time and memory budgets (i.e. time spent by a task is proportional to the energy consumed and the memory used is proportional to the memory energy consumption).

Another use of time and memory budgets would be to use this combination of guidelines to guide the compiler in the process of deciding what to optimize for, thus improving the compiled result.
Chapter 4

Validation

4.1 Introduction

To evaluate how useful the proposed BICD method is in the real case, experiments were done that evaluated two aspects of the method. One aspect is the scalability and optimality of the method. In other words, how close to an optimal solution the method comes in a certain amount of time, and how this scales with the size of the task graph. The other aspect is the usability, at implementation, of the results generated in the proposed method. Is it possible to reduce the implementation costs by avoiding unnecessary implementations and iterations?

4.2 Practicality

Since the proposed budget generation method is meant to be a part of a practical software tool, it is important that budgets that are good enough to use are generated in a reasonable time. Reasonable time in this context probably ranges from seconds, in the case of the designer who wants a quick estimate, to the length of a coffee break in the case of the engineer who wants a draft of the system to perhaps as much as ten to 16 hours in the case of the designer of the almost finished system. What is the definition of good enough budgets then? This depends of course on how far the project has been developed, but, ultimately, on how much knowledge has been gathered of the system. In a system where we have full knowledge
CHAPTER 4. VALIDATION

about the outcome of all implementations (i.e. all estimates have an uncertainty of 0), a good enough budget is one where the tightness is less than 1. On the other hand, if the system that is about to be synthesized has a great number of tasks with high degrees of uncertainty, then a good result would be a budget that maximizes the probability that implementation within the ICTs will be possible (i.e. aiming to minimize the tightness).

4.2.1 Method

The method for ITC derivation is intended to be used in a part of a system development process as a tool that can be used before and during the implementation of software components. Thus it is important that the method can be implemented in a way that makes it practical to use (i.e. has a reasonable response time) and that the ITCs produced will be close to optimal and will support the implementation process. This section therefore focuses on the evaluation of the tool’s practicality and optimality. The main concern as regards the practicality of the method is the fact that finding optimal ITC derivations involves an extensive search that grows rapidly as the size and complexity of the system grow.

The evaluation was made using a prototype Java implementation of the BICD. This means that there is potential for further improvements of the tool’s efficiency. The Java Virtual Machine used was the Java HotSpot(TM) Client VM (build 1.4.0-b92). The platform used has been single nodes in a Linux powered cluster. The nodes were configured as follows:

CPU  AMD 2000 (1666 MHz)
System bus  266 MHz
Memory  256 Mbyte DDR
Network interface card  Intel Ethernet Pro 100
Local disk  30 Gbyte, ATA-100

The evaluation consists of experiments based on example cases found in the literatures as well as generated cases. Details on the case structure and examples is found in Appendix [B]. Using this prototype implementation and the set of test cases, we addressed the following three questions:
4.2. **PRACTICALITY**

- What is the optimal tightness for each case? In many cases, the search space has been so large that we have had to settle for approximations based on the best solutions found during a very long run.

- Is our method practical? This means determining how close to an optimal solution it is possible to come within a time that is short enough to make the tool useful as a provider of quick feedback to a system designer about the feasibility of a certain design choice.

- How well do different search strategies perform?

### 4.2.2 Experiment

The purpose of these tests has been to prove the practicality of the BICD method. For this reason the collection of examples should reflect a wide variety of typical problem types and problem sizes. A more detailed description of the cases used for this evaluation is found in [21]. The evaluation is limited to platforms with homogeneous processors connected by a single bus. All tasks are assumed to be of the same periodicity and to be single instances.

Since in most cases it takes too long to find the optimal tightness for a case, we settled for an approximation of the optimum. The approximation is based on the best tightness found during a long run (i.e. ten hours). We denote this value as the "tightness limit", $t_{\min}$. Alternatively, it is also possible to calculate a lower limit for the tightness. The limit, denoted as the "theoretical tightness limit", $t_{\lim}$, is calculated as the tightness for a case in which all the tasks are allocated to separate processors, but without the addition of communication tasks (i.e. maximum parallelism). The problem with $t_{\lim}$ is that it is most likely that it will be impossible to reach this tightness in reality, even if it is sometimes possible. If $t_{\min}$ is compared to $t_{\lim}$, as in Figure 4.1, 50% of the cases come closer than 5% to $t_{\lim}$ and 75% of the cases come closer than 10% to $t_{\lim}$. The worst case is approximately 50% off. The fact that $t_{\min}$ comes close to $t_{\lim}$ is equal to the finding of a solution close to optimal. For a case that does not come close to $t_{\lim}$, this decision is not possible to do. The result may be close to the optimum although the tightness is far from the theoretical limit.

To be able to efficiently search through the enormous search spaces, several search heuristics are proposed and compared. The aim of this work has not been to find an optimal search method but to find a search method that is good enough to use to
Figure 4.1: The theoretical tightness limit, $t_{lim}$, for the different cases. The values are normalized to the minimum tightness, $t_{min}$, in the long experiments (i.e. the estimated optimal tightness), for each case.

prove the method useful. The following three search heuristics with the following different properties were evaluated.

**Linear** A linear search is done by selecting one schedule after another. Eventually (in very long time), all possible schedules have been evaluated. This strategy will work well for the cases where good budget solutions are evenly distributed in small clusters. The number of good solutions is not very important as long as the good solutions are evenly distributed.

**Random** Random search makes as many point hits in the search space as possible during the search. This results in many independent solutions. This strategy works well for the cases in which the good budget solutions are grouped in large clusters. The relative occurrence of good solutions is more important than the distribution of the solutions.

**MLinear** Multi-linear search is really several linear searches, all with different start points. In this specific evaluation pruning is applied to the group of searches in which the search is carried out for a certain period of time. The worst solutions identified up to that time are pruned, giving more time for the other, more promising, searches. This strategy is good for both cases described in the Linear and Random strategy descriptions.
4.2. PRACTICALITY

The experiments are divided into three run-time classes. These three classes are the following.

**Long** Long experiments that are allowed to run for ten hours.

**Short** Short experiments that are allowed to run for one hour.

**UShort** Ultra-short experiments that are allowed to run for five minutes.

The two run-time classes of long and short contain experiments for all three search heuristics: linear search, multi-linear search with random start points and pruning, and multiple random single shot search. For the UShort experiment, only the random method is used and is repeated nine times. The reason for repeating the short experiments is that the random searches will give different results for each execution, and this will be especially distinct in the short simulations. A summary of the results is shown in the Table 4.1.

A comparison of the long experiments for the three different search heuristics, see Figure 4.2, shows that they give similar results. There are a few bad cases for each of the heuristics, but no heuristic is significantly worse than another. For the short experiments, see Figure 4.3, the random search heuristic shows a slightly better result than the other two. Finally, when the random experiments are compared for all three experiment lengths, see Figure 4.4, it turns out that the ultra-short random search heuristic is about as good as the long random search.

The time it takes to find a useful budget could be defined as the time it takes to find a budget with a tightness of less than 5% from the optimal tightness, \( t_{\text{min}} \). As time in seconds can vary significantly from computer to computer, depending on the performance, the number of generated budgets and the number of milliseconds spent are measured to make the results easier to compare. This gives a performance index for each particular experiment. The expected result, of the evaluation of the number of budgets that was generated before 95% \( t_{\text{min}} \) is reached, was that more budgets would be required for larger systems. As seen in Figure 4.5, no distinct trend could be detected in the data from the experiments. Graph size is certainly a parameter since there is, in all cases but one, a lower bound that increases exponentially with the graph size, but there are several large systems that require the same number of budgets as the small systems in order to come close to \( t_{\text{min}} \).
Figure 4.2: The results of the ten-hour experiments using all the different search methods. The values are normalized to the minimum tightness, $t_{min}$, found in the three long experiments (i.e. the estimated optimal tightness), for each case. The linear and multi-linear search method did not find a feasible solution for the reconfigurable case.

Figure 4.3: The results of the one-hour experiments, using all the different search methods. The values are normalized to the minimum tightness, $t_{min}$, in the long experiments (i.e. the estimated optimal tightness), for each case. The linear and multi-linear search method did not find a feasible solution for the reconfigurable case.
Figure 4.4: The results of the experiments using the random search method for different durations. The values are normalized to the minimum tightness, \( t_{\text{min}} \), in the long experiments (i.e. the estimated optimal tightness), for each case.

Figure 4.5: The diagram shows how many iterations are needed to make the budgets reach within 5% of \( t_{\text{min}} \).
Figure 4.6: The deviation from $t_{\text{min}}$ for short random searches does not show a clear trend, as we had expected. This is suggested to be examined further in future work by adding a number of cases of greater size.

4.2.3 Discussion

For the BICD method to be of real interest, it must be proven that it can be used successfully for large systems with many tasks and complex constraints and that this scales well with increasing system size. A substantial set of cases has been used, ranging from five tasks to over 80 tasks. The results presented show that short runs are in many cases as efficient as long runs and that random searches are significantly better than linear searches. However, if the deviation from $t_{\text{min}}$ for short random runs is plotted against the task graph size (see Figure 4.6), defined as the sum of the number of tasks and edges, no clear trend can be found. The expected trend would be that the tightness for short runs would deviate more for increasing system size. The results indicate that short random searches result in reliable budgets in a short time. Similarly, it is shown that the number of budgets needed to come close to $t_{\text{min}}$ is not solely dependent on the task graph size. Since no trends can be detected, we suggest that, in future work, the set of experiments is extended and some other approximation of complexity than task graph size is defined.
4.3 Usefulness

Evaluation of the practical usefulness of the implementation constraints generated must be done by some kind of implementation of the tasks, where different implementation methodologies can be compared, some that do not use derived ICTs and some in which the ICTs can be taken into account and used to support the implementation. This section introduces a framework for this kind of evaluation and the subsection following gives specific details about the approach of achieving this by means of simulated implementation and about the experiments that used this approach.

The evaluation compares the implementations made using budgets and the implementations made using the *ad hoc* method. It would be optimal to make this evaluation in real life for real projects, but this would also be resource demanding. To get a quick estimate of whether the budget-based method is better than an *ad hoc* implementation method, we chose to simulate the implementation procedure. Simulation also makes it possible to obtain a large amount of statistical material and to repeat experiments for different implementation methods.

The simulation model used for the implementation simulator (see Figure 4.7) starts with information about the tasks from the design (TD). This information is used as input to the budgeting algorithm (Bx) which in turn generates imple-
CHAPTER 4. VALIDATION

mentation time budgets. Available budgets are forwarded to the (simulated) implementation (I) where the implementor tries to implement all tasks according to the calculated budgets, fulfilling at least one budget. Simulated implementation could be described as imagining a box with a lid for each task. The WCET for the implementation and the associated cost for this implementation are in this box. The outcome of the implementation is not known until the lid is removed, and, when the lid is removed you have to take the cost of the implementation. Implementations are evaluated by a schedulability analysis (SA) tool to decide whether it is possible to create an execution schedule for the set of implemented tasks. If this is the case, the simulation terminates positively (OK), signaling that there exists an implementation at a certain implementation cost. If no complete implementation exists according to the schedulability analysis, we have to reiterate, trying to modify the budgeting parameters. If no changes (no new implementations) have been made since the last budget generation, further budget generation is pointless and the simulation terminates negatively, signaling FAILURE.

When the budget-based implementation method is to be evaluated the budgeting method Bx is the BICD using the extension for estimates of different certainty. The budgeting algorithm generates several different budgets to avoid iterations if implementation according to some of the budgets fails. After budget generation, implementation will be simulated by implementing task by task in order of decreasing tightness. After each task implementation, budgets that are not fulfilled are removed and the process is repeated. If all budgets have failed, new budgets must be generated. It may be possible to find additional budgets since some knowledge about the system has been added by implementation. However, if all budgets fail without a single implementation being made this means that no new budgets will be found and FAILURE will be signaled. If an implementation is completed according to any given budget, this indicates that there exists at least one schedule for the implemented tasks, and OK will be reported. Otherwise, the partial implementation will be updated and used to set the parameters for a new set of budgets that may make implementation possible. If no alteration is possible, FAILURE will be reported.

When an ad hoc implementation is evaluated, the initial budgeting method, Bx, is a method that generates infinite budgets for all tasks. This means that all possible implementations are valid during implementation, which also means that we know that we will never need to re-budget. Implementation is done in the same way as described earlier, but, this time, a schedulability analysis is used to evaluate the implementation since we do not yet know whether we have found a schedulable implementation. If that is the case, OK will be reported. Otherwise, adjustments must be made. If adjustments are not possible (i.e. there are no further imple-
4.3. USEFULNESS

It is assumed that the implementation of a single task starts with a coarse proposal that is refined during the implementation into the final implemented component. To model this in simulation, it is assumed that all tasks have an estimate of the execution time for the component to be implemented (EWCET) and an actual worst case execution time (AWCET). The AWCET is the WCET for the implemented component and is unknown until implementation has been completed. After implementation, the cost in work hours is summed to give the total number of work hours.

In the real case, the result of the implementation could be that the AWCET violates the AET for the task, thus indicating task implementation failure. In the real world, the implementation is iterated, at additional cost, until the implementation succeeds to comply with the constraints or a maximum allowed cost is reached. This is modelled as an exponential function that determines the cost of decreasing the AWCET down to the AET. It is possible that the AWCET deviates so much from the AET that the function indicates that the cost of fixing this will be too large. If so, this results in task implementation failure and may be solved in two ways: (i) generating new ITCs or (ii) redesign.

AWCET generation

As described above, each task must have an AWCET and an associated cost in work hours. For each task, a random AWCET is drawn from a rectangular distribution centered at the EWCET, where the width of this window is proportional to the certainty of the estimate. To avoid trivial implementations, a constant can be added to the EWCET to shift the distribution, resulting in higher AWCETs and therefore implementation simulations of increased difficulty (i.e. introducing an offset for optimistic estimates). The AWCETs are drawn in the experiments from a distribution that is 1.5 times the estimate in width, and with an offset that sets the lowest AWCET to 5/8 of the estimate. For example, for a task with an estimate of 100, the offset $AWCET_o$ is 62.5 and the width of the distribution is 150, meaning that the AWCET will be between 62.5 and 212.5 (i.e. $AWCET_o$ and $AWCET_o + 1.5e$, respectively).
CHAPTER 4. VALIDATION

The cost model

The cost for a task’s implementation is considered as an inverse proportional function of the AWCET, where a lower AWCET leads to higher cost. This is implemented in the experiments as the estimate, $e$, divided by the AWCET minus the offset $AWCET_o$, plus a constant, $c$ (to avoid division by zero):

$$cost = \frac{e}{AWCET - AWCET_o + c}$$  \hspace{1cm} (4.1)

This is based on the assumption that the cost for implementing a certain task will be higher the closer to the $AWCET$ limit ($AWCET_o$) the implementation gets. The described cost function works for the single implementation alternative evaluation described in Subsection 4.3.2. For incremental implementations, however, a more detailed cost function will be needed.

4.3.2 Experiment

The evaluation has been made for the special case where only one implementation alternative per task is used. In this case, where each task has only a single implementation alternative, it is easy to calculate the cost of the ad hoc method since the cost for this special case is equal to the sum of the cost for the implementation of all tasks. Schedulability is evaluated and FAILURE or SUCCESS is reported. For the budget-based method, tasks must be implemented until all tasks are implemented or all newly generated budgets are violated. Updates of estimates and uncertainties are made and new budgets are generated. If no further implementations are possible, although new budgets are generated, FAILURE is signaled. If all tasks are successfully implemented, SUCCESS is reported.

The evaluation of the budget-based implementation method was made in comparison to the evaluation of the ad hoc implementation method. The methods were evaluated using the same set of applications and the same sets of budgets for each application. As stated above, the applications are a mix of handmade applications and applications generated by random, ranging in size from five to 80 tasks. The evaluation shows that the budget-based implementation costs less in terms of implementation work than the ad hoc implementation. This means that, in the case where implementation is impossible, the budget-based method indicates this at an early stage in the implementation (corresponding to a low cost), while the ad hoc method have to implement all tasks to even be able to evaluate the implementation
4.3. USEFULNESS

Figure 4.8: The relative cost of using the budget-based method drops for systems of larger size.

success (more costly). Note that, for the single implementation alternative experiments, the function used for calculation of the implementation cost is not critical. The choice of cost function will only affect the total cost for an implementation, not whether it will FAIL or not.

4.3.3 Discussion

Evaluation of the implementation simulation, using a single implementation alternative for each task, shows that the budget-based method always performs better than or equal to the *ad hoc* method for all system sizes (see Figure 4.8). As the figure shows, the relative cost (cost in work hours of the budget-based method divided by the cost in work hours of the *ad hoc* method) decreases for the budget-based method for cases of increasing size. A logarithmic regression has been added for clarity.

However, instead of evaluating the results as one group, the experiments can be divided into two groups, one for experiments resulting in FAILURE and one for experiments resulting in SUCCESS. In the cases of implementation failure, the budgeting method identifies this at a lower cost in work hours than the *ad hoc*
method (see Figure 4.9). A logarithmic regression has been added for clarity. In the case of successful implementation, the cost of using the budgeting method will be the same in work hours as in the ad hoc method.

These results clearly point out that the proposed budget-based method will be cheaper when it is needed most – when the system is about to break down. The proposed method should be extended further, allowing better modelling of implementation cost and supporting incremental implementations. It would also be interesting to evaluate other popular implementation methods for comparisons.

To make the implementation simulation more realistic, a possible extension would be to allow several possible implementations alternatives and not just one. One possibility would be to assign initial costs and AWCETs (e.g. according to the single alternative method described earlier) and extend this with the possibility to make minor adjustments at a low cost. One way would be to model incremental implementation as a function in which an input of the desired AWCET results in an AWCET and an associated cost. The cost could be exponentially proportional to the difference between the old AWCET and the desired AWCET. Incremental implementation would be possible down to a predetermined level of AWCET, below which the cost becomes infinite.
4.3. USEFULNESS

of communication overhead and resource interference.

In the maximization allowed number of processors, it may not be possible to get a tightness equal or even close to $t_{lim}$ because $t_{lin}$ indicates the minimal achieved tightness. Value $|V|$ indicates the number of tasks in the task graph, and value $P_{max}$ is the maximum allowed number of processors. It may not be possible to get a tightness equal, or even close, to $t_{lim}$ because of communication overhead and resource interference.

Table 4.1: Summary of the results of the experiments, showing the relative sizes of the test cases and the tightness of the best budgets found. "lin" stands for linear search, "mlin" stands for multi-linear search, and "rnd" stands for random search. The tests are divided into three categories based on their execution time. Value $t_{lim}$ indicates the absolute limit for the tightness, and $t_{min}$ indicates the minimal achieved tightness. Value $|V|$ indicates the number of tasks in the task graph, and value $P_{max}$ is the maximum allowed number of processors.

| Case      | $t_{lim}$ | $t_{min}$ | $|V|$ | $P_{max}$ |
|-----------|-----------|-----------|------|-----------|
| Neural    | 0.678     | 0.699     | 8    | 10        |
| Simple    | 0.438     | 0.451     | 15   | 10        |
| Radar     | 0.501     | 0.501     | 12   | 10        |
| Combination | 0.760   | 0.814     | 12   | 4         |
| Simple PA | 0.760     | 0.814     | 12   | 4         |
| Simple PAC| 0.760     | 0.814     | 12   | 4         |
| Simple UA | 0.760     | 0.781     | 15   | 4         |
| Neural    | 0.678     | 0.699     | 8    | 10        |
| Simple    | 0.438     | 0.451     | 15   | 10        |
| Radar     | 0.501     | 0.501     | 12   | 10        |
| Combination | 0.760   | 0.814     | 12   | 4         |
| Simple PA | 0.760     | 0.814     | 12   | 4         |
| Simple PAC| 0.760     | 0.814     | 12   | 4         |
Chapter 5

Conclusions and Future Work

Handling non functional requirements is poorly addressed by most software development methodologies today. This will turn out to be a real problem when system complexity increases since a faulty design will be detected very late in the development process, probably after implementations are finished. There is a clear need for a method that is able to keep development costs caused by inflexible designs to a minimum. The ICD method proposed in this thesis has the potential to be a useful practical and cost-saving approach that handles non functional requirements in embedded software development in a better way than most present competitors.

The ICD method reported here handles non functional constraints that make implementation a much clearer task, introducing implementation guidelines (ITCs). Although the problem of finding optimal budgets may seem intractable at first, the experiments show that close to optimal budgets can be found in a very short time (i.e. a matter of minutes). In most the cases it has been shown that the budgets are actual near optimal; for the rest of the cases it has been shown that no significantly better budgets can be found in a long period of time (i.e. ten hours).

Experiments using the budgeting method for implementation have shown that the use of ITCs reduces implementation costs for complex distributed real-time computer systems. The experiments show that great reductions in implementation cost can be achieved for projects that are border on breakdown. It is even possible to detect which projects are about to collapse merely by looking at the ITCs available during the implementation. This makes it possible to make an early statement about what costs the development team could expect for different design propos-
Although several experiments have been carried out with positive results, it is important to further validate the ICD method using documented, practical (i.e. real) cases. The current potential of the method is described in Chapter 3, where, for example, extensions for task graphs with tasks of different periodicity and support for several independent constraint descriptions for the same task set (i.e. multi-case) are presented. It is also important to identify how the method performs when the cases are switched from generated to real cases. Based on these future experiments the question of how the ICD method could, and perhaps should, be used will be answered.

Use of the budgets from the ICD method will be further examined with the purpose to evaluate how the budgets could be used in real software development projects. Further experiments will show whether the extended models are suitable for parameters that will be used in the practical cases or whether even further extensions are needed. One important question is, for example, how helpful the budgets will be since, they could be based on unreliable estimates. It would most probably not be a problem since the budgets are supposed to be used in an iterative way, generating new budgets based on updated estimates as soon as additional information has been gained. Further, if all estimates are of the same uncertainty it does not really matter that the estimates have a low certainty since it will still be possible to make trade-offs (i.e. the budgets will even out). A verification of these assumptions, through an evaluation of the importance of the precision of the estimates, would be good.

Future work will also be to explore how the non functional constraints depend on the functional constraints and the functional decomposition, and how to fit the ICD method to present development methods. Further extensions that should be evaluated are how to extend the communication support to include other types of communication networks (e.g. multiple busses) and how to model multi-hop communication. When a project comes to a dead end, is it really possible to simply switch to another budget as is proposed in the method? This can be investigated through large-scale experiments where the method is put to work with real projects and real people.


References


Appendix A

Related Work

A.1 Introduction

There exists several tools and methods that solve parts of the constraint derivation problem. However, most of them focus on the implementation validation and scheduling, and not design and design verification. Research focusing on design verification does not normally treat the possibility of handling multiple resources at the same time (e.g. time and processor allocation). In the cases where multiple resources are considered, no guarantees are given for the implementation probability or even implementation possibility.

Summaries of a number of different research projects similar to the one presented in this thesis are given below.

A.1.1 Deriving internal timing constraints

There are methods that in some sense try to achieve or achieves results similar to the results presented in this thesis (i.e. derivation of internal timing constraints). One of these related methods uses a view of high-level timing issues in the design and validation of embedded real-time systems [2]. The derivation problem is defined as the problem of deriving internal timing constraints from external timing constraints in an embedded real-time system. The requirements specification
phase describes what the system’s external behavior is, without specifying how the system works internally. The internal view is described in the architectural design phase in terms of tasks.

It is stated that the derivation problem the designers try to solve is: "Given the system’s external timing constraints and task structure, derive its internal timing constraints such that the satisfaction of the internal timing constraints implies the satisfaction of the external timing constraints from which they have been derived, and validate the remaining external constraints."

Another observation is that the current practice for handling of the timing problem is based on "trial and error" guided by engineering experience. Typically the designer will first of all focus on designing a functionally correct system and then try to address the non-functional constraints. This is most likely due to the fact that neither design methods nor real-time programming languages used for design of real-time systems provide sufficient support for timing constraints.

A.1.2 A UML-based design methodology for real-time and embedded systems

The increasing complexity of current real-time embedded systems necessitates new design methods and tools to face the problems of design, analysis, integration and validation of complex systems \[3\]. Integration and validation of complex systems becomes a bottleneck in the traditional design flow. Full separation of hardware and software cannot longer be sustained.

This paper mainly address the control, communication and synchronization refinement of model and constraints during the conceptual design stages. It is pointed out that specifications written in human language (e.g. english) have a number of shortcomings. Specifications are written in natural language and are simply ambiguous. Typically these informal, incomplete specifications are directly transformed into real-time level descriptions.

A design method for real-time and embedded systems is proposed, which make use of system level specification that describes the system in its real environment by means of use cases and context diagrams. However, it would be hard to use this method in really early design stages since formal specifications are assumed.
A.1.3 Guaranteeing real-time requirements with resource-based calibration of periodic processes

A solution for the problem of component based derivation of end-to-end requirements of real-time systems is proposed in [9]. The main goals are to automate the design process in some sense and to minimize processor utilization. The proposed solution is also able to handle cases where all end-to-end constraints cannot be fulfilled and some transformations are needed.

An example of an iterative ad-hoc process for design is shown and it is pointed out that the high level of freedom is a problem when trying to make correct decisions. A requirement derivation method is therefore presented, where the system model for the proposed solution has very high freedom and describes all timing parameters as relative values such as jitter and skew. A severe restriction for the method is that just single processor systems are handled. In later papers the authors propose solutions for multiple processor systems.

Although the authors say that this is a design method, it seems more likely that it is useful as an implementation method since the proposed method assumes all tasks’ maximum execution times to be known in advance.

A.1.4 Dynamic end-to-end guarantees in distributed real-time systems

The objective of the paper [4] is to present a scheme that allows dynamic scheduling and guaranteeing of distributed processes communicating via synchronous primitives. The method is divided into two parts, online and offline. The offline method is based on an optimization metric that looks for paths with minimum laxity per computation time. This optimization metric is motivated as "The normalized laxity metric seems appropriate if we consider that lengthy tasks are likely to be more difficult to schedule."

When the offline method has divided the task graph into slices, grouping tasks with the same normalized laxity, these slice are sent to the online scheduler that tries to schedule them (i.e. there is no guarantee that the system will be schedulable).
A.1.5 The slack method: a new method for static allocation of hard real-time tasks

A constructive heuristic for the allocation of periodic hard real-time tasks to multiprocessor or distributed systems, called the slack method, has been evaluated [1]. As pointed out in the title of this paper, the method addressed is an allocation method and as such it assumes the knowledge of for example worst case execution time for each task. The allocation should be optimized in a way that allows us to use a minimum number of processors (resources) at the same time as requirements are satisfied. The proposed solution is to use critical path clustering for the allocation. The method is divided into two steps, (i) find a feasible assignment and (ii) limit the resource use.

The authors state that "The main weakness of the slack method is that it sometimes fails in finding any feasible assignment, although there is one, due to the fact that the graph reductions lead to loss of information, or due to communication interference which is intentionally ignored.".

A.1.6 Resource conscious design of distributed real-time systems: an end-to-end approach

"We model the problem as a constraint solving problem, in which the original end-to-end timing constraints are expressed as a set of constraints on task attributes." is stated in [15]. This work is the extension of the original solution, which was limited to single processor systems. Starting from a given task graph, and a set of end-to-end constraints, task attributes are systematically generated such that (i) the task set is schedulable, and (ii) the end-to-end constraints are satisfied. This involves deriving task periods, deadlines, phases, and synthesizing code for inter-task communication.

The main contribution of this work is the systematic methodology to transform a high-level design into a schedulable system. This provides the designer with a rapid prototyping tool and helps the designer to fix and optimize a faulty design for both correctness and performance.
A.1.7 Experiences from introducing state-of-the-art real-time techniques in the automotive industry

The authors of [6] draw the conclusion that the use of the proposed methods increase the time spent in design, but shortens the implementation time/cycle. The use of state-of-the-art real-time techniques in industry remains infrequent. The reason for this, as believed by the authors, is three-fold: (i) the lack of commercially available tools, (ii) the lack of methodologies based on real-time theory throughout the complete development process, and (iii) the lack of competence in real-time theory among industrial practitioners.

It is stated that development of complex computer systems is an expanding field and that many of these systems also could qualify as real-time systems. The increased complexity of these systems leads to increasing demands with respect to requirements engineering, high level design, early error detection, productivity, integration, verification, and maintenance. This calls for methods, models and tools that permit a controlled and structured development during the complete life cycle of the system. The proposed method and model have been validated using an extensive case study of a single industrial project.
APPENDIX A. RELATED WORK
Appendix B

Detailed Case Descriptions

The purpose of the experiments presented in this thesis have been to prove the practicality and usability of the BICD algorithm. The collection of cases used in the experiments should reflect a wide variety of typical problem types and problem sizes. The cases presented in this thesis are limited to systems with deadlines shorter than the period. The information in the case descriptions are limited to precedence constraints, allocation constraints, end-to-end timing constraints and the estimated execution time. As representative examples, a list of a few of the different cases and their properties is given below.

**Slack**  This case, called The Slack Method [1], is a very small problem with a very tight solution.

**BStat**  This case, called Base Station [23], describes a very simple regulator.

**Small**  This case, called Small [5], has just one solution and is very tight. This problem contains mirrored solutions (i.e. that should be avoided for efficiency reasons).

**Reconf**  This case, called Reconfigurable [17], is a quite large example. It is shown that neither EDF nor RM finds a solution for this problem.

**SafeC**  This case, called Safety Critical [14], is a system with redundant tasks.

**Tind**  This case, called Tindell [1], is a typical regulator for a mechatronical system.
APPENDIX B. DETAILED CASE DESCRIPTIONS

**MReg** This case, called Multiple Regulator, is developed by the author of this thesis. The case is quite small but still quite hard. The problem is composed of three separate regulators.

**SCtrl** This case, called Safety Control [12], is a typical regulator, quite easy to find solutions for.

**Comb** This case, called Combination, is a combination of the two cases SCtrl and SReg. Since these cases are easily solved separately in short time, a combination should also be quite easy to find solutions for.

**RadarM** This case, called Radar Matrix [7], describes the computation chain for an active radar system (i.e. an actively controlled array of antennas).

**SReg** This case, called Simple Regulator, is developed by the author. The case describes a quite simple regulator but very hard to solve.

**NNet** This case, called Neural Network, is developed by the author of this thesis. The case is really small and is easy to solve (mainly used for testing).

Below follows a summary of the primitives used to construct the case descriptions:

**PROC_MAX** The maximum allowed number of processors.

**NODE** Each node in the task graph. Each node has a name, estimated execution time, release time, and deadline. Release time and deadline are optional where "not defined" is indicated by -1.

**EDGE** Each precedence constraint in the task graph. Each edge has a source task and one or several destination tasks.

**COMM** For each precedence constraint there could optionally exist a communication cost. The communication cost has a source task, a destination task, and an estimated communication time. A corresponding communication task will be generated if the source task and the destination task are allocated on separate processors.

**GROUP** It is possible to group tasks in clusters, applying locality constraints. This is defined as a list of tasks that have to be allocated on the same processor.
UNGROUP A task has also the possibility to avoid being grouped with specific tasks. This is defined as a list of tasks, where no task is allowed to share processor with any other task in the list.

FIX Tasks have the possibility to fix allocation to a single processor. This is defined as the task and the processor number.

To be able to keep different versions of the same task graphs an additional notation was introduced. A short summary is given below.

FA Fully allocated, all tasks are allocated (i.e. fixed).
PA Partly allocated, one or several tasks are allocated.
UA Un-allocated, no task is allocated.
xxC FAC, PAC and UAC, the postfix C indicates that this is a condensed version of the task graph (i.e. it has fewer nodes).

For example purposes, the following sections consist of selected printouts of interesting task graph descriptions, used in the experiments. Complete descriptions of all cases are found in [21].
B.1 slackUA

# Slack UA
#

PROC_MAX 5
#

Define nodes with command Node
#

<table>
<thead>
<tr>
<th>Name</th>
<th>Est</th>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>N0</td>
<td>4</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>N1</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N2</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N3</td>
<td>1</td>
<td>-1</td>
<td>18</td>
</tr>
<tr>
<td>N4</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N5</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N6</td>
<td>6</td>
<td>-1</td>
<td>18</td>
</tr>
<tr>
<td>N7</td>
<td>4</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
B.1. SLACKUA

Node N8, 6, -1, 18
Node N9, 4, 0, 18

# Define edges with command Edge
# Src, Dst1, Dst2, ..., DstN
Edge N0, N2, N1
Edge N1, N3, N5, N4
Edge N2, N3
Edge N4, N6
Edge N5, N6
Edge N7, N8

# Define communication tasks with command Comm
# Src, Dst, Est
Comm N0, N2, 1
Comm N0, N1, 1
Comm N1, N3, 1
Comm N1, N5, 1
Comm N1, N4, 1
Comm N2, N3, 1
Comm N4, N6, 1
Comm N5, N6, 1
Comm N7, N8, 1
B.2  baseStationPA

# BStat PA
#

PROC_MAX 3

# Define nodes with command Node
#
# Name,  Est,  Start,  Stop

```
```
Node T1, 9600, 0, -1
Node T2, 78225, -1, -1
Node T3, 73975, -1, -1
Node T4, 9675, -1, -1
Node T5, 40750, -1, -1
Node T6, 23400, -1, 1000000
Node T7, 5400, 700000, -1
Node T8, 11500, -1, -1
Node T9, 5400, 700000, -1
Node T10, 17125, 0, -1
Node T11, 28450, -1, -1
Node T12, 89050, -1, -1
Node T13, 17625, -1, -1
Node T14, 71725, 200000, -1
Node T15, 8200, -1, -1
Node T16, 122500, -1, -1
Node T17, 73975, -1, -1

# Define edges with command Edge
# Src, Dst1, Dst2, ..., DstN
Edge T1, T2
Edge T2, T3
Edge T3, T4
Edge T4, T5
Edge T5, T6
Edge T7, T8
Edge T8, T4
Edge T9, T4
Edge T10, T11
APPENDIX B. DETAILED CASE DESCRIPTIONS

<table>
<thead>
<tr>
<th>Edge</th>
<th>Src, Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge T11, T12</td>
<td></td>
</tr>
<tr>
<td>Edge T12, T13</td>
<td></td>
</tr>
<tr>
<td>Edge T13, T4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edge</th>
<th>Src, Dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge T14, T15</td>
<td></td>
</tr>
<tr>
<td>Edge T15, T16</td>
<td></td>
</tr>
<tr>
<td>Edge T16, T17</td>
<td></td>
</tr>
<tr>
<td>Edge T17, T4</td>
<td></td>
</tr>
</tbody>
</table>

# Define communication tasks with command Comm
# Src, Dst, Est

<table>
<thead>
<tr>
<th>Comm</th>
<th>Src, Dst, Est</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm T8, T4, 50</td>
<td></td>
</tr>
<tr>
<td>Comm T9, T4, 25</td>
<td></td>
</tr>
<tr>
<td>Comm T14, T15, 3775</td>
<td></td>
</tr>
<tr>
<td>Comm T15, T16, 6500</td>
<td></td>
</tr>
<tr>
<td>Comm T16, T17, 650</td>
<td></td>
</tr>
<tr>
<td>Comm T17, T4, 650</td>
<td></td>
</tr>
<tr>
<td>Comm T13, T4, 650</td>
<td></td>
</tr>
</tbody>
</table>

# Fix nodes with command Fix
# Node, Resource

<table>
<thead>
<tr>
<th>Fix</th>
<th>Node, Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix T1, 1</td>
<td></td>
</tr>
<tr>
<td>Fix T6, 1</td>
<td></td>
</tr>
<tr>
<td>Fix T7, 2</td>
<td></td>
</tr>
<tr>
<td>Fix T9, 2</td>
<td></td>
</tr>
<tr>
<td>Fix T10, 3</td>
<td></td>
</tr>
<tr>
<td>Fix T14, 3</td>
<td></td>
</tr>
</tbody>
</table>
B.3. BASESTATIONPAC

B.3  baseStationPAC

#  BStat PAC
#

PROC_MAX  3

#  Define nodes with command Node
#
#  Name,  Est,  Start,  Stop

<table>
<thead>
<tr>
<th>Node</th>
<th>Name</th>
<th>Est,</th>
<th>Start,</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>161800</td>
<td>0,</td>
<td>0,</td>
<td>-1</td>
</tr>
<tr>
<td>T2</td>
<td>73825</td>
<td>-1,</td>
<td>900000,</td>
<td>-1</td>
</tr>
<tr>
<td>T3</td>
<td>71725</td>
<td>200000,</td>
<td>-1,</td>
<td>-1</td>
</tr>
<tr>
<td>T4</td>
<td>130700</td>
<td>-1,</td>
<td>-1,</td>
<td>-1</td>
</tr>
<tr>
<td>T5</td>
<td>73975</td>
<td>-1,</td>
<td>-1,</td>
<td>-1</td>
</tr>
<tr>
<td>T6</td>
<td>152250</td>
<td>0,</td>
<td>700000,</td>
<td>-1</td>
</tr>
<tr>
<td>T7</td>
<td>16900</td>
<td>700000,</td>
<td>-1,</td>
<td>-1</td>
</tr>
<tr>
<td>T8</td>
<td>5400</td>
<td>700000,</td>
<td>-1,</td>
<td>-1</td>
</tr>
</tbody>
</table>
# Define edges with command Edge
# 
# Src, Dst1, Dst2, ..., DstN

Edge T1, T2
Edge T3, T4
Edge T4, T5
Edge T5, T2
Edge T6, T2
Edge T7, T2
Edge T8, T2

# Define communication tasks with command Comm
#
# Src, Dst, Est

Comm T1, T2, 500
Comm T3, T4, 3775
Comm T4, T5, 6500
Comm T5, T2, 650
Comm T6, T2, 650
Comm T7, T2, 50
Comm T8, T2, 25

# Group nodes with command Group
#
# Node1, Node2, Node3, ..., NodeN

Group T1, T6

# Spread nodes with command Ungroup
#
# Node1, Node2, Node3, ..., NodeN

Ungroup T1, T5
Ungroup T2, T5

# Fix nodes with command Fix
#
# Node, Resource
Fix $T_1$, 1
Fix $T_6$, 1
B.4 smallUA

PROC_MAX 2

# Define nodes with command Node
# Name, Est, Start, Stop

Node A, 18, 0, 36
Node B, 10, 0, 10
Node C, 15, 0, 25
Node D, 17, 10, 35
Node E, 10, 25, 35
# SReg PA
#

# Define nodes with command Node
#
#
<table>
<thead>
<tr>
<th>Name</th>
<th>Est,</th>
<th>Start,</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>T1,</td>
<td>5,</td>
<td>25,</td>
</tr>
<tr>
<td>Node</td>
<td>T2,</td>
<td>23,</td>
<td>-1,</td>
</tr>
<tr>
<td>Node</td>
<td>T3,</td>
<td>7,</td>
<td>-1,</td>
</tr>
<tr>
<td>Node</td>
<td>T4,</td>
<td>5,</td>
<td>0,</td>
</tr>
<tr>
<td>Node</td>
<td>T5,</td>
<td>12,</td>
<td>-1,</td>
</tr>
<tr>
<td>Node</td>
<td>T6,</td>
<td>41,</td>
<td>-1,</td>
</tr>
<tr>
<td>Node</td>
<td>T7,</td>
<td>7,</td>
<td>-1,</td>
</tr>
<tr>
<td>Node</td>
<td>T8,</td>
<td>5,</td>
<td>0,</td>
</tr>
</tbody>
</table>
APPENDIX B. DETAILED CASE DESCRIPTIONS

<table>
<thead>
<tr>
<th>Node</th>
<th>T9</th>
<th>33</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>T10</td>
<td>7</td>
<td>-1</td>
<td>75</td>
</tr>
<tr>
<td>Node</td>
<td>T11</td>
<td>5</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Node</td>
<td>T12</td>
<td>12</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Node</td>
<td>T13</td>
<td>8</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>Node</td>
<td>T14</td>
<td>16</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Node</td>
<td>T15</td>
<td>5</td>
<td>-1</td>
<td>250</td>
</tr>
</tbody>
</table>

# Define edges with command Edge
# Src, Dst1, Dst2, ..., DstN

<table>
<thead>
<tr>
<th>Edge</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>Edge</td>
<td>T4</td>
<td>T5</td>
</tr>
<tr>
<td>Edge</td>
<td>T5</td>
<td>T6</td>
</tr>
<tr>
<td>Edge</td>
<td>T6</td>
<td>T7</td>
</tr>
<tr>
<td>Edge</td>
<td>T8</td>
<td>T6</td>
</tr>
<tr>
<td>Edge</td>
<td>T9</td>
<td>T10</td>
</tr>
<tr>
<td>Edge</td>
<td>T11</td>
<td>T12</td>
</tr>
<tr>
<td>Edge</td>
<td>T12</td>
<td>T9</td>
</tr>
<tr>
<td>Edge</td>
<td>T13</td>
<td>T14</td>
</tr>
<tr>
<td>Edge</td>
<td>T14</td>
<td>T15</td>
</tr>
</tbody>
</table>

# Define communication tasks with command Comm
# Src, Dst, Est

<table>
<thead>
<tr>
<th>Comm</th>
<th>T1</th>
<th>T2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm</td>
<td>T2</td>
<td>T3</td>
<td>1</td>
</tr>
<tr>
<td>Comm</td>
<td>T4</td>
<td>T5</td>
<td>2</td>
</tr>
<tr>
<td>Comm</td>
<td>T5</td>
<td>T6</td>
<td>1</td>
</tr>
<tr>
<td>Comm</td>
<td>T6</td>
<td>T3</td>
<td>1</td>
</tr>
</tbody>
</table>
B.5. SIMPLEREGPA

Comm T6, T7, 1
Comm T8, T6, 1
Comm T8, T9, 1
Comm T9, T7, 1
Comm T9, T10, 1
Comm T11, T12, 2
Comm T12, T9, 1
Comm T13, T14, 1
Comm T14, T15, 3

# Fix nodes with command Fix
#
# Node, Resource
Fix T1, 1
Fix T3, 1
Fix T4, 2
Fix T7, 2
Fix T8, 2
Fix T10, 2
Fix T11, 3
Fix T13, 4
Fix T15, 4
B.6 neuralNetUA

```
# NNet UA
#

# Define nodes with command Node
# Name, Est, Start, Stop
#
Node T1, 50, 0, 400
Node T2, 100, 50, 400
Node T3, 50, -1, 400
Node T4, 35, -1, 400
Node T5, 25, -1, 400
Node T6, 20, -1, 400
Node T7, 40, -1, 400
Node T8, 35, -1, 400
```

# Define edges with command Edge
#
# Src, Dst1, Dst2, ..., DstN

Edge T1, T3
Edge T2, T3
Edge T3, T4, T5
Edge T4, T6, T7
Edge T5, T7, T8

# Define communication tasks with command Comm
#
# Src, Dst, Est

Comm T1, T3, 10
Comm T2, T3, 10
Comm T3, T4, 10
Comm T3, T5, 10
Comm T4, T6, 10
Comm T4, T7, 10
Comm T5, T7, 10
Comm T5, T8, 10