Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors

Koch, Peter

Publication date: 1996

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

? Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
? You may not further distribute the material or use it for any profit-making activity or commercial gain
? You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.
Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors

Peter Koch
The DSP Research Group
Institute for Electronic Systems
Aalborg University
Fredrik Bajers Vej 7, B2
DK-9220 Aalborg, Denmark

December, 1995
Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors

PhD Dissertation¹ by Peter Koch, MSc. EE

¹The work is conducted under an ordinary PhD grant from Aalborg Universitets Center, 1990
Acknowledgements

First of all I'll like to express my sincere gratitude to Dr. Kallol Bagchi, Stanford University, CA., and Professor Kjeld Hermansen, Aalborg University, to whom I am in great debit for many valuable discussions, proposals, and solid support throughout my studies.

A special thank goes to Professor Ole Olsen, Aalborg University, who took away so many hours of my ordinary teaching load paving the way for a deep and thorough study.

Also, I should thank Dr. Anders Færgemand Høyer, TeleDanmark Research, for his many outstanding questions and suggestions, and Mr. Hans Roelsgaard Jensen, L. M. Ericsson A/S, for his comments on the Simulated Annealing Scheduler.

Furthermore, I am grateful for all the support which I have received from my family when this work required all my attention for long periods of time.
Abstract

The ever increasing complexity of Digital Signal Processing and other data independent scientific computations means that single processor systems in many cases do not provide the required computational capacity in order to respect given time constraints. Architectures with multiple DSP processors are therefore becoming popular alternatives. This raises, however, a natural requirement for formal methods which can help the system designer to create realistic and efficient multiprocessor solutions. Of particular importance in this design process is static multiprocessor scheduling which 1) assigns nodes (i.e., subalgorithms) to the individual processors, 2) specifies the node execution order, and 3) on the basis of estimated execution and communication times calculates invocation times for the individual nodes.

We have found no or very little reported work on 1) the overall formal design trajectory, 2) realistic strategies for static multiprocessor scheduling of DSP algorithms, and 3) guide lines which may help the designer selecting an appropriate scheduling strategy.

We therefore present an overall design trajectory which specifies the interrelation between the various tasks required in the design process. Our particular interest is the multiprocessor scheduling problem which is a much neglected research area. Previously proposed scheduling strategies fail to take into account a realistic InterProcessor Communication (IPC) cost which in most cases affect the final multiprocessor solution significantly. As a consequence, we therefore develop realistic models for IPC handling in shared memory and message passing architectures.

Based on these models, we next devise several new scheduling techniques, basically to be classified as either 1) list scheduling heuristics, or 2) Simulated Annealing (SA) based scheduling strategies. Belonging to the former class are our Hu*, NODUST/ASAP, and NODUST/CP heuristics. Hu* is an IPC-improved version of an existing method, and the NODUST algorithms are both techniques which employ the principle of node duplication. Moreover, we included in our repertoire of list scheduling heuristics the Dynamic Level Scheduler (DLS) which has recently been proposed by researchers from University of California, Berkeley. This method has been proclaimed as the best list scheduler known today.

We next devise the SAS strategy which is our fundamental SA based scheduler. It has, as compared to the list scheduling heuristics, the outstanding feature of providing search in the total solution space thus increasing the probability of finding the optimal schedule. Due to the iterative nature of SAS, we found, however, a significantly run time increase as compared to the list scheduling heuristics. In order to relax this problem, we next develop SAGUM, a SAS based strategy which utilizes the potential IPC overhead into the individual nodes as a guide mechanism for the search procedure. Beside, to further reduce the run time overhead, we created a hybrid ILst SA scheduler, LISA, which initiates the SA search from the near-optimal DLS solution.

From a series of experiments with 1) randomly generated DSP algorithms (in terms of precedence graphs), and 2) off-the-shelf DSP algorithms, we conclude that for i) algorithms with limited inherent parallelism, or ii) a small IPC overhead, the Hu* heuristic is superior. In other case, the DLS approach seems best, the exception being for very high IPC overhead where the NODST/CP heuristic always generates better solutions. If one can accept the run time penalty, we found that our SA based strategies with almost no exception outperform all the list scheduling heuristics. SAGUM seems superior in terms of the solution quality, and LISA is the strategy which generates a solution using the smallest run time overhead.

From the numerous experimental results, we finally extract a set of guide lines which in detail describe how to select one or more of the strategies in order to generate efficiently a realistic multiprocessor solution.
Dansk Resumé


Vi har i den tilgængelige videnskabelige litteratur ikke været i stand til at finde udtømmende beskrivelser af hverken 1) et overordnet formel designforløb, 2) realistiske strategier for statisk multiprocessor-scheduling af DSP-algoritmer, eller 3) retningslinier, som kan benyttes i forbindelse med udvælgelse af en anvendelig schedulersstrategi.

Vi præsenterer derfor beskrivelsen af en overordnet formel designmetode i form af en diskussion omkring de nødvendige trin, der skal gennemløbes. I den forbindelse koncentrerer vores interesse sig primært om multiprocessor scheduleringsproblemet, som er et meget forsonligt forskningsområde. De hidtidig foreslåede strategier tager ikke i forhold til omfang hensyn til den uundgåelige kumugnationstid mellem processorerne (IPC), hvilket i langt de fleste tilfælde har en negativ indflydelse på den endelige løsning. Med udgangspunkt i denne kendsgerning har vi udviklet modeller, som på realistisk vis håndterer IPC-begræb i arkitekturer baseret på henholdsvis fælles memory og punkt-tiplunkt kommunikation.

På baggrund af disse modeller har vi udviklet en række nye schedulersnings-teknikker, som kan grupperes som enten 1) list schedulerings-heuristiker, eller 2) strategier, der er baseret på Simulated Annealing (SA). Vores Hu*, NODUST/ASAP og NODUST/CP er alle heuristiker, som tilhører førstnævnte kategori. Hu* er en IPC-optimeret version af en eksisterende heuristic, mens begge NODUST-algoritmerne baserer sig på principippet om eksekvering af de samme nodes på flere processor (dvs. duplicering). I vores repertoire af list schedulerings-heuristiker har vi desuden inkluderet Dynamic Level Scheduling (DLS) metoden, som forskere fra University of California, Berkeley, for nylig har udviklet. Metode er i den videnskabelige litteratur blevet udråbt som den til dato bedste list schedulerings-heuristik.


På baggrund af en lang række scheduleringseksperimenter udført med 1) tilfeldigt genererede DSP-algoritmer (i form af precedence-grafer), og 2) off-the-shelf DSP-algoritmer kan vi konkludere, at for i) DSP algoritmer med en begrænset iboende parallellitet, eller ii) et lille IPC-forbrug, udviser vores Hu* heuristic den bedste performance. I andre tilfælde tyder resul-
taterne på, at DLS metoden genererer de bedste løsninger. En vigtig undtagelse er dog i tilfælde
med meget stor IPC, hvor resultaterne viser, at NODUST/CP heuristikken er bedst. Såfremt
system-designeren er villig til at acceptere den forholdsvis lange køretid, som karakteriserer de
SA-baserede strategier, kan vi konkludere, at disse metoder i langt de fleste tilfælde leverer bedre
løsninger i sammenligning med samtlige list schedulerings-heuristikker. Sammenlignes de tre SA-
metoder, synes SASGUM at generere de bedste løsningen, mens LISA er den strategi, som har
det mindste køretidsforbrug.

Med udgangspunkt i de eksperimentielle resultater har vi som afslutning ekstraheret en
mængde performancemæssige karakteristika. Disse er efterfølgende blevet brugt til udarbejdelse
af retningslinier for hvordan, der udvælges en eller flere schedulerings-strategier med henblik på
effektivt at tilvejebringe en realistisk multi-DSP løsning.
# Contents

1 Introduction

  1.1 A Design Trajectory for Multi-PDSP Solutions .................................. 3
  1.1.1 Specification and Algorithm Selection ........................................ 4
  1.1.2 Deriving an Architectural Model ............................................. 4
  1.1.3 Precedence Graph Generation .................................................. 5
  1.1.4 Adding Architectural Information onto the DAPG ............................. 5
  1.1.5 Scheduling the DAPG onto the Target Architecture ......................... 5
  1.1.6 Evaluation of the Generated Schedule ...................................... 7
  1.1.7 Optimization by Grain Size Adjustment .................................... 7
  1.2 The Goal of Our Work .................................................................... 7
  1.3 The Organization of The Thesis ................................................... 10

2 Scheduling and Models

  2.1 Fundamentals for Architectural Models ......................................... 12
  2.2 Several Ways to Model the Computations ....................................... 14
  2.3 Formal Scheduling Notations ...................................................... 14
  2.4 A SM IPC-Model with Improved Accuracy ...................................... 17
  2.5 An IPC Model for PDSP Message Passing ....................................... 24
  2.6 A Note on Synchronization Mechanisms ....................................... 29

3 List Scheduling Heuristics

  3.1 The Scheduling Technique Proposed by Hu ..................................... 31
  3.2 An Improved Hu Algorithm ......................................................... 33
  3.3 A Hu Inspired IPC Heuristic ...................................................... 35
    3.3.1 Realistic Scheduling with the Hu* Concept ................................. 36
    3.3.2 Is the Processor Selection Procedure Optimal? .......................... 39
  3.4 Scheduling by the Means of Node Duplication ................................ 39
    3.4.1 The Working of NODUST/ASAP ............................................. 40
    3.4.2 The Working of NODUST/CP ............................................... 45
    3.4.3 The Application of NODUST ............................................... 45
  3.5 Comparing Hu* and NODUST Using Random Graphs ............................ 46
    3.5.1 Graph Topologies ............................................................... 46
    3.5.2 Discussions on the Communication Overheads ............................ 48
    3.5.3 The Impact of Changing CCR-values ..................................... 51
    3.5.4 The Impact of Changing Graph Parallelism ............................... 52
  3.6 Dynamic Level Scheduling ....................................................... 53
  3.7 Scheduling In the Presence of Heavy Communication .......................... 56
4 Static Scheduling using Simulated Annealing
   4.1 Fundamentals of Simulated Annealing ........................................ 59
       4.1.1 The Metropolis Criterion .............................................. 60
       4.1.2 Essential Elements in SA ............................................. 61
   4.2 Pioneers in SA Based Scheduling ........................................... 63
   4.3 SAS — The Simulated Annealing Scheduler .................................. 65
       4.3.1 Moving The Selected Node ........................................... 66
       4.3.2 From Processor Entry to Schedule .................................... 67
       4.3.3 Deadlock — A Possible Situation .................................... 69
       4.3.4 Employing the Metropolis Criterion ................................ 71
       4.3.5 Choosing an Appropriate Cooling Strategy .......................... 72
       4.3.6 Creating an Initial Assignment/Ordering ............................ 73
       4.3.7 The Overall SAS Strategy .............................................. 74
   4.4 Setting the SAS parameters .................................................. 75
       4.4.1 Length of the initial transition sequence ............................ 75
       4.4.2 Distance parameter and Markov chain length ....................... 77
       4.4.3 The stop criterion ...................................................... 81
   4.5 Performance Evaluation of SAS .............................................. 83
       4.5.1 Choosing the SAS Parameters ........................................ 83
       4.5.2 Results of The SAS Experiments .................................... 84
   4.6 Using SAS In the Presence of Heavy Communication ..................... 86

5 Suggestions for SAS Redesign .................................................. 88
   5.1 Transition Functions with Guidance ....................................... 88
       5.1.1 Optimizing In and Out Communication ............................... 89
       5.1.2 Performance Evaluation of SASGUM ................................. 90
   5.2 Combining List and SA Scheduling ......................................... 95
       5.2.1 Matching c to a Near-Optimal Solution ............................. 97

6 Case Studies ............................................................................. 102
   6.1 Selected Off-the-Shelf DSP Algorithms .................................... 103
       6.1.1 The Durbin Algorithm ................................................... 104
       6.1.2 The Cholesky Factorization ............................................ 105
       6.1.3 The Block Based Autocorrelation .................................... 107
       6.1.4 The Recursive Autocorrelation ....................................... 108
       6.1.5 The Fast Fourier Transform .......................................... 109
       6.1.6 The Wave Digital Filter ................................................ 110
   6.2 An ADSP-2106x Based Target Architecture ................................. 111
   6.3 The Scheduling Results ....................................................... 114

7 Conclusion and Perspective ..................................................... 121
   7.1 A Review of Our Work .......................................................... 121
   7.2 A Set of Scheduling Guide Lines .......................................... 124
   7.3 A Perspective View ............................................................. 126
## CONTENTS

### A Preprocessing the DSP Algorithms

- **A.1 Signal Flow and Data Flow Graphs** .................................................. 133
  - A.1.1 The Signal Flow Graph Notation .................................................. 134
  - A.1.2 The Data Flow Graph Notation .................................................. 134
- **A.2 The Synchronous Data Flow Notation** ............................................. 135
- **A.3 An Algorithm for PASS Generation** ............................................... 139
- **A.4 Constructing an DAPG** ............................................................... 140
  - A.4.1 From SDFG to DAPG — A Case .................................................. 143

### B Randomly Generated DAP Graphs

- **B.1 Specification of Parameters within the GGT** .................................. 148
- **B.2 Algorithms for Graph Generation** .................................................. 149
  - B.2.1 Linear Decreasing Probability .................................................. 150
  - B.2.2 Reciprocal Decreasing Probability ........................................... 152
- **B.3 Representing the DAP Graphs** ..................................................... 153
- **B.4 Defining the Randomly Generated Test Graphs** .............................. 153

### C Published Papers

- **C.1 Paper 1** ......................................................................................... 155
- **C.2 Paper 2** ......................................................................................... 155
- **C.3 Paper 3** ......................................................................................... 156
- **C.4 Paper 4** ......................................................................................... 156
Chapter 1

Introduction

In recent years Digital Signal Processing (DSP) has become a very popular alternative to traditional analog processing principles in almost all kinds of electronic system design. There is in particular one reason for that evolution; the extreme progress made in microelectronic technologies. Since the first microprocessor, the Intel 4004 [1], appeared in 1972, single chip devices, e.g., for numerical computations, have shown 1) a complexity increase in about a factor of 2 annually, 2) a constant improvement in their operational speed of around 20% per year, and 3) a reduction in the cost by an annual factor of between 1.6 and 2 [2, 3].

These circumstances have paved the way for development of high-performance single-chip microcomputer types of architectures especially suited for algorithms with typical DSP characteristics. Such general purpose devices, which are known as Programmable Digital Signal Processors (henceforth to be denoted PDSP), have been commercially available since the late 1970s [4, 5]. We note that PDSPs in general are significantly different in their architectural style as compared to e.g., standard microprocessors. In particular, the PDSP is targeting towards a broad spectrum of applications requiring extensive arithmetic computation, usually with hard real-time constraints. Consequently, the typical PDSP architecture and its instruction set are not designed with operating system support in mind.

In order to further optimize the match between a given DSP algorithm and the underlying target architecture, numerous researchers have addressed the domain of Application Specific Integrated Circuits (ASIC), i.e., special purpose (usually non-programmable) architectures optimized for one and only one DSP application algorithm, e.g., [6, 7, 8]. This approach is appealing when simultaneous constraints on 1) chip area, 2) execution time, 3) power consumption, and 4) numerical properties are part of the overall design specification. Because of the ever increasing demand for more processing capability required by the DSP community (due to the continuously growing algorithmic complexity), this design approach is frequently employed to fulfill the computational requirements.

However, the complexity of the ASIC design procedure is magnitudes higher than its PDSP counterpart. Therefore, because of the ever decreasing "time to the market" requirement, researchers at an early stage realized the necessity for formal design methods in order to create automatic ASIC design tools. Substantial research in this field has been carried out during the last decade, and various systems for Data Path- and Controller Synthesis\(^1\) have been constructed, examples are [9, 10, 11, 12, 13, 14, 15, 16]. These systems typically take as an input an executable behavioral specification formulated in terms of an applicative language, e.g., the DSP-oriented Silage language [17]. Based on the behavioral description, as well as on various hints from the

\(^1\)Normally referred to as High-Level Synthesis.
CHAPTER 1. INTRODUCTION

designer (primarily concerning types and numbers of functional building blocks to be applied in the target architecture), the tools generate a register transfer (RT) description which can next be applied by module generators which produce the layout for the integrated circuit.

Unfortunately, only very few of these systems have actually ever reached the commercial state. One exception though, is the CATHEDRAL-I Silicon Compiler [18, 19] for bit-serial digital filter design which in 1993 was made commercial as The DSP-Station [20] through Mentor Graphics, European Development Center, Belgium. Because the CATHEDRAL-I project was originally launched way back in the early 1980s, this indeed tells us something about the significant amount of time required to develop and commercialize an ASIC design tool.

In this perspective, alternative methods capable to achieve the performance specifications required by state-of-the-art DSP applications have to been searched for. Turning our attention back to the PDSP, one may suggest increased clock frequencies as well as more efficient data path and controller architectures in order to improve the single-chip performance. However, it is a fact that state-of-the-art processors are already employing highly efficient sub-micron technologies, and therefore it seems very difficult to obtain any progress in terms of increased switching capability. Furthermore, modern PDSPs are typically featuring what is known as micro parallelism which essentially means that several on-chip execution units are able to operate concurrently by the use of Very Long Instruction Words. This facility is made possible by numerous on-chip busses allocated for simultaneous instruction- and data moves. Architectural tuning of the individual processors is therefore also an approach which hardly can provide the substantial processing overhead required by numerous applications.

In spite of these facts, the most immediate possibility for improvements is therefore to employ multiple cooperating processors in a parallel architecture. Previously, manufactures of PDSPs have provided on their devices only some very sparse interprocessor connection facilities but this situation is now changing. Various manufactures, e.g., Texas Instruments [21], Motorola [22], and Analog Devices [23], have recently launched PDSPs with multiple specialized communication features which make these devices especially qualified as building blocks for multiprocessor architectures.

However, despite that multiprocessing PDSP devices are now becoming commercially available, the most significant object for designing applications using multiple processors can be formulated very concise;

- How is the application partitioned into minor simultaneously executable subtasks and how are these tasks distributed optimally onto the underlying multi-PDSP architecture?

As we will outline in the following, the transformation of an initial algorithmic description into an optimal multiprocessor assembly code actually consists of numerous non-trivial tasks which indeed makes it very difficult to give a clear and straight forward answer to the question.

This can also be recognized from the fact that only few researchers up to this point of time have reported on formal methods for mapping DSP applications onto architectures incorporating multiple PDSPs. One may argue that the absence of substantial research results in this domain is due to the fact that PDSPs previously have not been designed with multiprocessing in mind. This, of course, is true but, on the other hand, there are no immediate hindrances on most previous PDSPs which disqualify them as multiprocessors building blocks.

We may therefore conclude that in order to fulfill the computational requirements demanded by many state-of-the-art DSP applications, an architecture with multiple off-the-shelf PDSPs may be an appropriate solution. The required design procedure, however, turns out to be non-trivial and we therefore need a rigorous design trajectory in order to obtain efficient solutions. To our knowledge, such a trajectory has not been previously reported in the literature.
CHAPTER 1. INTRODUCTION

In the following we therefore in brief discuss the kind of tasks naturally required to be included in such a design scheme.

1.1 A Design Trajectory for Multi-PDSP Solutions

From the experiences reported by various researchers who have designed and implemented design schemes for High-Level Synthesis (HLS), it is a well-known fact that a large number of individual tasks are required in the overall mapping procedure. Also, from the HLS domain it is known that a completely automatic design environment is very difficult to construct. Furthermore — and perhaps most important — a "push-the-button" system will apparently not be able to generate solutions which are as optimal as hand-crafted solutions. Basically, this is due to the fact that formal methods seems to be very difficult to formulate from "designer experiences".

Therefore, the design scheme that we suggest is denoted interactive because it performs some of the hard and tedious work, and next provides the result to the designer who is then responsible for 1) making a qualitative evaluation, and 2) the decision on what to do next.

Now, in order to get an idea of tasks to be included into our scheme, we first take a look at tasks which are typically employed in HLS tools, [24]. Basically, they are

- support for generation of an executable specification
- precedence graph generation, i.e., textural to graphic transformation
- allocation of execution units (EXU), e.g., ALUs, multipliers, etc.
- assignment of arithmetic operations onto the EXUs
- RT code generation

Similarly, a mapping procedure for multi-PDSP target architectures must incorporate several tasks. First of all, the need for a task to extract and expose the inherent parallelism in the algorithm is obvious. The extraction can be done by parsing the executable specification into a graphic representation in terms of a precedence graph. The question just is "at what level" the parallelism has to be extracted? Should it be at the atomic level where we handle single arithmetic operations or should it be at a higher level where multiple arithmetic operations are clustered into larger sub-algorithms? Basically, this is a matter of adjusting the sizes of the sub-algorithms to fit the target architecture. In the literature, this is known as the grain size problem, [25]. Adjusting the grain size may be implemented by some kind of clustering/declustering technique.

In order to solve the grain size problem, it is evident that an à priori knowledge of the actual runtime of each individual sub-algorithm is needed and thus a method for extracting architectural information is next required.

Moreover, scheduling is a very essential task required in a multi-PDSP design scheme. Scheduling (and in this context we mean multiprocessor scheduling) may mean different things to different people. In a multicomputer environment (e.g., a network of workstations) the aim of scheduling is to provide fair share of processing resources to user jobs (multiprogramming and time sharing) [26]. Another situation occurs when the target architecture is composed of cooperating (signal) processors like in our case. Here, the purpose of scheduling is to distribute a single application/algorithm (which is initially partitioned into several sub-algorithms as discussed above) onto the processors in order to optimize a given objective function (OF). Examples of OFs are 1) schedule length, i.e., the overall execution time (henceforth makespan), 2) load balancing,
3) the cost of communication between processors, or 4) a weighted combination of these. In general, the problem of finding an optimal schedule (for any OF) is a **combinatorial optimization problem** which is known to be NP-hard [27]. In most cases, **exhaustive search** is therefore out of scope and **scheduling heuristics** are normally needed. This emphasizes why the problem of mapping algorithms onto multiple interconnected PDSPs cannot be characterized by a simple and unambiguous solution.

Now, tools for code generation are also required in order to create a full working design scheme. The algorithmic descriptions have to be compiled into machine code for the individual processors. Furthermore, appropriate synchronization and communication primitives (if required by the architecture) have to be generated and implemented on the processors.

Moreover, advanced tasks like loop-folding, and life time analysis of variable (for minimization of memory requirements), may be considered in order to generate solutions comparable in performance with hand-crafted assembly programs.

By now it should be clear that the overall procedure required for implementation of parallel PDSP systems is far more complicated as compared to the situation of a mono-processor architecture. We therefore believe in the necessity of a rigorous trajectory in order to design and evaluate systematically various design alternatives.

In the following we therefore present a proposal for such a scheme (as illustrated in figure 1.1) in terms of a brief description of the individual tasks, their purposes and interrelation.

### 1.1.1 Specification and Algorithm Selection

Like most other design schemes, our method initially requires a **functional description** of the actual application. This description is created in terms of an **executable behavioral specification**. Additionally, we require a definition of the constraints within which the application has to be executed, i.e., information on maximum processing time (sample interval), the required computational accuracy, the maximal physical size, maximal power consumption, and so on. These informations are provided as an independent **specification description**.

Based on the application- and specification descriptions, an **initial algorithm selection** for the application is performed. This selection procedure is based on high-level simulations of numerous algorithm candidates. The purpose being a comparison of the algorithms in terms of e.g., numerical properties. Test vectors for this comparison are supplied from the application description.

Essentially, the output from the algorithm selection procedure is a set of algorithms which can be successfully employed for the application. It should be noted that the proposed structure do not prohibit an **algorithm design phase** as an alternative to the selection procedure. It may turn out that off-the-shelf algorithms do not suit the specifications and thus the system designer is enforced to create his own algorithm for the particular problem.

### 1.1.2 Deriving an Architectural Model

The target architecture is composed out of processor residing in the Processor Pool. Basically, this is a collection of various PDSPs which are suitable for multiprocessor configurations. Initially, we assume that 1) the target architecture can be **constructed** on the basis of the information provided from the specification description through the application of a **processor selection procedure** [28], or 2) the multiprocessor architecture is **given** prior to the actual design phase.

Now, a model of the target architecture is required in order to construct realistic multiprocessor solutions. An important task in our scheme is therefore the **architectural modeling** task, where basic informations about the architecture are extracted. This concerns essentially informations
related to execution time for various operations and the time for communication of partial results between processors (e.g., in terms of estimated numbers of clock cycles). A detailed knowledge about the processor instruction set and the synchronization mechanisms among the processors is therefore a necessity.

1.1.3 Precedence Graph Generation

Prior to scheduling of the algorithms on the target architecture, the algorithms must be pre-processed in order to bring them onto a form suitable as input for the scheduling procedures. In appendix A we explain in some detail a proper input format; the Directed Acyclic Precedence Graph (DAPG). Obtaining this format requires that a transformation from a textural to a graphic form can be performed. In this work we will not address the problems of textural specification to precedence graph transformation, but instead we will assume that a Data Flow Graph representing the algorithm is available.

Lee and Messerschmitt, [29], proposed the data flow notation known as Synchronous Data Flow (SDF) which is especially suited for handling DSP and other types of data independent algorithms. The SDF notation supports efficiently DSP-algorithms because it relies on an à priori knowledge of the number of samples (henceforth tokens) consumed and produced, respectively, by every node in the graph in every sample interval.

In DSP types of algorithms, there are typically very few data dependent operations and the number of tokens produced and consumed can therefore in most cases be determined prior to execution. In general, however, DSP algorithms may contain data dependent operations. For those algorithms where data dependent operations are required, special care must be taken in order to specify worst case values for the number of tokens produced and consumed. Similarly, worst case figures for estimated run- and communication times are required in such situations.

We also note that the SDF notation can handle a broad range of grain sizes. Depending on the initial algorithm partitioning, the granularity can be atomic, medium, or large. This is an important property since the actual grain size has a fundamental impact on the overall performance of the final multiprocessor solution.

Since an SDF graph (SDFG) may contain cycles, the precedence relations may not be obvious and therefore the graph has to be made acyclic by the means of a SDFG-to-DAPG transformation.

1.1.4 Adding Architectural Information onto the DAPG

Once the DAPG is derived, the graph is extended according to the information provided by the architectural model. Basically, the estimates for execution- and communication times are added to the graph which thus inherits the properties of the target architecture. The DAPG is now ready to be scheduled.

1.1.5 Scheduling the DAPG onto the Target Architecture

The Multiprocessor Scheduling Pool encloses a number of scheduling strategies characterized by different properties, e.g., 1) search complexity, compile time, communication handling, etc.. The reason for this multiplicity is argued by the fact that no researchers to our knowledge have reported on guide lines for the selection of a specific scheduling strategy given 1) a precedence graph (characterized by e.g., interconnection topology, degree of inherent parallelism, and ratio between execution and communication times), and 2) a target architecture. We therefore propose the facility to experiments with different strategies.

Normally, people are referring to one of two possible types of scheduling,
CHAPTER 1. INTRODUCTION

- dynamic scheduling
- static scheduling

In the case of dynamic scheduling, the sub-algorithms are assigned to a specific processor during runtime, i.e., by a runtime scheduler which is part of the underlying operating system. Static scheduling, on the other hand, is performed at compile time which means that the operating system (if any) do not handle the scheduling task. According to [30], static scheduling may be classified due to the taxonomy

- static allocation
- self-timed scheduling
- fully static scheduling

Basically, static allocation assumes the existence of a local run-time scheduler which invokes sub-algorithms that are allocated to the individual processors at compile time. A sub-algorithm cannot be invoked before all its input data are available, which means that the execution order is unknown at compile time since no information on execution and communication times are provided.

By fixing the execution order at compile time, the run time scheduling overhead may decrease and the self-timed approach can be employed. Here a simple synchronization mechanism ensures that the next sub-algorithm scheduled for execution is initiated whenever all its input data are ready. Determination of the execution order may be due to some estimated values of the execution- and communication times.

Finally, if we know exactly 1) the execution time of each sub-algorithm, and 2) the time required for communication, then fully static scheduling may be employed. In such a case, the start time of each individual sub-algorithm can be calculated prior to the execution and thus the schedule is said to be fully deterministic. This approach, of course, is appealing but since it requires very detailed knowledge, the system designer needs to craft assembly code and next calculate or measure run- and communication times for all parts of the program before the schedule can be created.

It is therefore obvious, that static scheduling is a matter of finding a suitable trade-off between compile- and run time. Here we should clarify that static scheduling in the following means

- assignment of sub-algorithms onto the processors
- specification of the execution order of the sub-algorithms (on the individual processors)
- specification of the invocation times based on estimated execution times

and thus we for DSP systems consider static scheduling to be something in between self-timed scheduling and fully static scheduling.

Onwards, we assume that 1) our target architecture consists of multiple PDSPs, 2) and that the target applications are data independent algorithms, like most DSP algorithms and other scientific computations. In this perspective, we consider the dynamic scheduling approach as being out of scope. This is primarily due to the fact that hard real-time constraints normally eliminate the possibility for a substantial scheduling overhead at runtime.

Output from the scheduling task is an abstracted solution in terms of a Gantt Chart, i.e., a graphic representation of the schedule.
1.1.6 Evaluation of the Generated Schedule

The generated schedule must be evaluated in order to determine whether the specification description is fulfilled. However, it may only be a subset of the specifications that can be efficiently evaluated. The constraint typically placed on the sample rate is easily verified, whereas, e.g., the power consumption may not be straightforward to evaluate.

Due to the hard real-time constraints associated to most DSP application, it is reasonable to believe that the system designer is especially interested in the makespan obtained. Therefore, a reliable decision can be made whether to proceed to 1) code generation (in case the program can be executed within the given sample interval) or 2) to optimize the solution (in case the program fails to be executable within the given sample interval).

In the later case, there are several possibilities. As suggested previously, the grain size may not be optimal. Therefore, if the system designer is not immediately interested in experiments with other scheduling strategies from the pool, the grain size of the DAPG may alternatively be adjusted.

1.1.7 Optimization by Grain Size Adjustment

The key point here is that the scheduling strategies residing in the pool generate solutions which take into account the time required to communicate tokens between processors. Due to this non-zero communication time, it is obvious that the smaller the grain size the higher the probability that a substantial part of the schedule is occupied with communication. In such a case it may therefore be an advantage to cluster the individual operations into larger grains and thereby reduce the potential communication time. On the other hand, this will also decrease the degree of available parallelism — which typically has the opposite effect, namely an increase of the makespan. Alternatively, it may be necessary to trade in a low communication overhead for a higher degree of parallelism. The DAPG, which has just been scheduled, is applied as input to the grain size adjust procedure and a new DAPG with modified grain sizes is generated. The modified graph is next scheduled using 1) the same scheduling procedure once more, or 2) another procedure from the pool.

In case the specification descriptions cannot be met by adjusting the grain size, we emphasize the alternatives;

- Redesign of the DSP algorithm(s), i.e., algorithm manipulation.
- Selection of another off-the-shelf algorithm with a similar functionality.

1.2 The Goal of Our Work

So far we have discussed the purposes of the individual tasks that we suggest for a multi PDSP design trajectory. However, in this work it is too ambitious to believe in a creation of the complete scheme. We are able to contribute only partial to the overall framework. Therefore, in the following, we describe our domain of interest and specify the purpose of this research work.

Over the last decades, many researchers, have been working in parallel computing but most of them have looked at multiprocessor systems such as PCs, workstations and mainframes interconnected through local or long distance networks which, due to the hard real-time constraint of DSP systems, is out of scope here.

As a matter of fact, we were initially not able to find much reported work within the domain of multi-PDSP system design. One particular paper by Lee and Messerschmitt [29], however,
caught our attention. To our knowledge, it was among the first papers which actually addressed formal design of DSP multiprocessing systems. The paper introduces some ideas for convenient description of DSP algorithm. Basically, Lee and Messerschmitt argue that DSP system designers naturally describe their algorithms in terms of block diagrams. We agree upon that as long as we are looking at the initial stages of the design trajectory. When it comes to the real implementation, a textural specification, e.g., VHDL code, has to be generated in order to synthesize software and/or hardware. Also, it’s a fact that many DSP design engineers are using a high-level language, such as Silage, C or MATLAB\(^2\), for algorithm design and/or simulation. It is reasonable to believe that a block diagram or equivalently a data flow graph has been created in order to support the programming phase.

Based on these observations, Lee & Messerschmitt adapt (partly from [31]) a special DSP optimized data flow notation which they entitle Synchronous Data Flow. The principle idea is based on the fact that (most) DSP algorithms are totally deterministic, i.e., that the number of tokens 1) produced and 2) consumed by every block upon each execution can be specified prior to the actual execution of the algorithm — therefore the name synchronous.

Due to this deterministic behavior, a mathematical formulation of an SDF graph can be devised. Now, DSP algorithms may 1) contain loops, and 2) have one or more blocks which require multiple executions within a single sample interval (i.e., multi sample-rate systems). It mean that the precedence relation among the individual blocks may not be immediately obvious. Therefore, [29] describes formal methods for 1) transformation of an SDF graph into an Periodic Admissible Sequential Schedule (PASS) — in order to prove the existence of a parallel schedule (the sequential schedule can be interpreted as a special case of a parallel schedule) — and 2) transformation of an SDF graph into a Directed Acyclic Precedence Graph (DAPG).

The authors claim that their method generate

- a PASS, and
- a PAPS (Periodic Admissible Parallel Schedule).

However, we do not in full agree upon the last point. Mapping an SDF graph into an PAPS requires an intermediate stage in terms of a precedence graph, and from our point of view, this is actually what the work of Lee & Messerschmitt is all about (see e.g., appendix A).

They basically devise a method for DAPG generation and their work is therefore (at the risk for us to be considered as being pedantic) not a complete trajectory which map a DSP algorithm in terms of an SDFG onto the final parallel schedule. Their method is an efficient technique for preprocessing DSP algorithms in order to prepare for multiprocessor scheduling.

Although the mathematical proofs for the PASS and DAPG generations, are thoroughly described, the actual algorithms for these purposes are unfortunately expressed in terms of incomplete pseudo codes from which we believe it is impossible to design an SDFG-to-DAPG converter. In order to solve this problem, our initial interest therefore was to devise a more clear specification of PASS and PAPS generation algorithms.

Next, “The missing tool” in Lee & Messerschmitt's work is basically a scheduling mechanism which maps the DAPG onto a parallel architecture, i.e., produces a parallel schedule. In order to argue that they are performing static scheduling of SDF graphs, they therefore (as a final stage in the “design trajectory”) employ a technique developed by T. C. Hu for assembly line scheduling way back in 1961, [32].

Despite the somewhat simplistic nature of this scheduling heuristic, it can be shown (under certain assumptions) that it is producing optimal solutions. One of those assumptions, however,

\(^2\)MATrix LABoratory by MathWorks, Inc.
is particularly severe — the InterProcessor Communication (IPC) times are assumed equal to zero. It means basically, that once a sub-algorithm (i.e., a node in the SDFG) has completed its execution, the node(s) which follow hereafter (i.e., successor(s)) can be invoked immediately on other processors. One may argue that in some situations it is a reasonable assumption — and we agree. When the average execution time of the individual blocks is much larger than the average communication time required (if two interconnected blocks are assigned to different processors), then a scheduling technique which neglects the communicational aspects (like the one proposed by Hu) may produce realistic and efficient solutions.

However, in many designs of parallel DSP systems we have reason to believe that the IPC time cannot be neglected — basically because 1) the granularity may be small, and 2) a realistic static schedule which obey the hard real-time specification requirements have to be generated.

Therefore, we found a need for solid techniques for static multi-PDSP scheduling which realistically take into account the IPC aspects. This is our primary domain of interest, and according to this, we formulated the following goals of the work;

---

Through the design of various realistic and efficient strategies for static multiprocessors scheduling, our purpose is to create a "Multiprocessor Scheduling Pool" which we have argued is a vital element in an efficient design trajectory for multi-PDSP system design. Next, through results obtained by making numerous scheduling experiments with these strategies, to formulate guide lines for the selection of a specific scheduling strategy given 1) a DSP algorithm with certain characteristics, and 2) an underlying target architecture.

---

To our knowledge, no researcher has previously suggested any guide lines to be used in DSP multiprocessing, and only very few researchers have actually reported on similar experimental environments. Known works are 1) PTOLEMY, [33], and 2) GRAPE, [34].

In short, PTOLEMY is an environment for heterogeneous simulation where different types of algorithms and hardware can co-exist — mainly for simulation purposes. Algorithms are specified in terms of a schematic entry paradigm where algorithm icons at different abstraction levels can be extracted from a number of different libraries and next composed into a complete system. As an experimental facility, PTOLEMY supports a simple multiprocessor scheduling environment (with only 1 scheduling heuristic) which may be employed on algorithms specified by the SDF notation. Code generation for the Motorola DSP56001 is supported.

GRAPE, on the other hand, is a CASE tool for the design of DSP algorithms and their implementation on a DSP multiprocessor (or ASIC). GRAPE supports the translations of specifications in C and Silage. The tool includes a partitioner which, from estimated execution and communication times, assigns the sub-algorithms onto the processors. Next, a scheduler is invoked in order to 1) finalize the ordering on the individual processors and 2) inserting communication primitives. GRAPE also supports DSP56001 code generation. From [34], however, it is unclear how the partitioner and the scheduler work.

In this thesis, we limit ourselves to non-massively homogeneous architecture, i.e., architectures with at most 6-8 processors of identical types. We note that other researchers have started to look into the field of heterogeneous processing, i.e., the situation were the architecture holds multiple processors of different types, [33]. Definitely, heterogeneous processing and other related techniques, like HW/SW Co-design, will gain tremendous attention in the years to come, but here we investigate only homogeneous architectures, basically due to the fact that from our point of view many questions concerning scheduling onto such architectures are still open.
1.3 The Organization of The Thesis

The thesis is organized as follows. Before we initiate the design of our scheduling strategies, chapter 2 gives a presentation of computational and architectural models. This is an important basis for our strategies since they are not designed to derive the actual multiprocessor programs, but rather abstracted solutions in terms of assignment, ordering and start time specification of the individual program modules. Chapter 2 also specifies 1) a formal scheduling notation according to which we design our strategies, and 2) various options for communications in shared memory and message passing architectures.

Chapter 3 is devoted to the design of list scheduling heuristics. We start by reviewing the technique which has provided a basis for the Lee & Messerschmitt trajectory — the Hu algorithm. Based on this strategy, we next propose some modifications in order to improve the reliability of the generated solutions, basically in terms of IPC handling. Through a search for techniques which may provide more efficient solutions, i.e., higher speedups, we propose node duplication in terms of our NODUST algorithm. Finally, we make some comparative studies between our strategies and a technique entitled Dynamic Level Scheduling (DLS), which recently has been devised and pointed out as the best known static list scheduling technique of today.

Due to the fact that list scheduling heuristics in most cases provide only near-optimal solutions, we launch in chapter 4 a study of more exhaustive search algorithms based on Simulated Annealing. After an introduction of the fundamentals, we cite previous work in this field, and then propose our Simulated Annealing Scheduler. A discussion on how its parameters are optimally tuned for our type of problems is next provided. We close this chapter with a performance evaluation where comparisons are made to the NODUST and the DLS heuristics.

It is evident that the more exhaustive a search technique, the more time is required before it converges. This is actually what characterizes the simulated annealing scheduler (as compared to the list scheduling heuristics). In chapter 5, we therefore suggest some modifications aiming at improving or maintaining the performance in terms of speedups while lowering or maintaining the computational complexity.

Next, in order to illustrate the working of our scheduling strategies, chapter 6 presents a series of scheduling experiments performed on off-the-shelf DSP algorithms. Finally, chapter 7 draws the conclusions, and put our work into perspective by suggesting numerous ideas for future work.

Additionally, three appendices are provided. The first one discusses our technique for data flow to precedence graph conversion which we have adapted from Lee & Messerschmitt.

In order to make a fair comparison between our scheduling techniques we need a set of test vectors with known characteristics. Therefore, we devised a graph generator tool which is presented in the second appendix.

The third appendix provides an overview of the papers that were written and presented at international conferences during the course of this thesis.
Figure 1.1: The proposed interactive multi-PDSP design scheme.
Chapter 2

Scheduling and Models

In order to provide the designer a flexible, reliable, and efficient framework for multi-PDSP design experimentation, generation of the exact assembly code program is out of the scope. Instead, we aim at abstracted solutions sufficiently accurate to guide the designer towards a qualified selection of a schedule which fulfill the initial specification descriptions.

To do so, the design of our Multiprocessor Scheduling Pool must rely on a formal notation upon which the individual strategies can be developed. Furthermore, we need models for 1) the target architecture and 2) the algorithmic computation. These subjects are elaborated in this chapter.

2.1 Fundamentals for Architectural Models

In chapter 1 we discussed the necessity for an architectural modeling task in order to generate realistic solutions. Basically, the scheduling strategies must have knowledge of the underlying architecture in terms of 1) the number of processors, 2) their interconnection topology, and 3) the IPC- and synchronization mechanism applied. Abstracted information on these parameters provides knowledge for an architectural model.

In the following, we therefore on the basis of the survey given in [35] adapt suitable architectural models. We note that the models discussed in [35] are devoted to multicomputer architectures which may not in full fit our type of target architectures. In a multicomputer architecture, it is assumed that the individual computers during execution are spending time in one of four states:

- performing calculations — $T_{Calc}$
- making message transfers — $T_{Comm}$
- doing housekeeping — $T_{House}$
- being idle — $T_{Idle}$

Based on this initial model, a further subdivision for all but the first state can be specified. For the example, $T_{Comm}$ can be more accurately expressed as:

$$T_{Comm}(p_i) = T_{Init}(p_i) + T_{Term}(p_i) + T_{Route}(p_i)$$  \hfill (2.1)
which means that the communication time associated to processor $p_i$ is described by three variables indicating the time required for computation associated with message *initiation*, *termination*, and *through-routing*, respectively.

Note that the individual terms cannot be associated to the same message transfer. Assuming a message which has to be transmitted from processor $p_i$ to processor $p_j$. This particular transfer is then described in terms of $T_{\text{Init}}(p_i)$, $T_{\text{Term}}(p_j)$, and $T_{\text{Route}}(p_k)$ where $p_k$ represents the processor(s) through which the message eventually has to be routed.

$T_{\text{Init}}(p_i)$ and $T_{\text{Term}}(p_j)$ represent the time associated to 1) copying data out of local memory, 2) generating packets, 3) function call overhead required for communication, 4) receipt of the packets, and 5) copying data into local memory.

$T_{\text{Route}}(p_i)$ represents the time required to transfer tokens from the input communication channel to the output communication channel on processor $p_i$. Therefore, as dedicated communication hardware evolve and is subsequently implemented as an integrated part of the processors, the $T_{\text{Route}}(p_i)$ overhead gradually will be superfluous.

Concerning the time required for *housekeeping*, this specifically includes three terms:

$$T_{\text{House}} = T_{\text{Sched}} + T_{\text{Inter}} + T_{\text{Recalc}}$$

(2.2)

In the case of dynamic scheduling, $T_{\text{Sched}}(p_i)$ represents the overhead imposed by a local run-time scheduler implemented on processor $p_i$. It may be very difficult in advance predicting the actual time the processor spends calculating the schedule. The primary reason is due to the fact that scheduling at run-time is influenced by the timing of the communication, which, on the other hand, depends on the actual scheduling.

The term $T_{\text{Inter}}(p_i)$ (*interference*) describes all the actions which are not associated to the scheduling task and which are only necessary on processor $p_i$ because it holds multiple communicating subalgorithms. An example is buffering of tokens which are arriving while a node is being executed.

In this model, $T_{\text{Recalc}}(p_i)$ describes a calculation overhead which may appear on processor $p_i$ if we allow part of the overall algorithm to be recalculated. In section 3.4 we discuss this opportunity and show that it may have a positive impact on the makespan. According to this description, the initial calculation of a subalgorithm is refered to as $T_{\text{Calc}}$ and recylcations performed at a later point in time are denoted by $T_{\text{Recalc}}$.

Finally, the model distinguish among two types of $T_{\text{Idle}}$; $T_{\text{Wait}}(p_i)$ and $T_{\text{Fin}}(p_i)$. The former is associated to idle time that may occur between execution of sequential tasks on processor $p_i$ whereas the latter refers to the period that processor $p_i$ stays idle after it has completed all its tasks. Therefore, $T_{\text{Fin}}$ equals zero for the processor(s) which execute the final task(s) in the schedule. It may be advantageous to divide $T_{\text{Wait}}(p_i)$ into $T_{\text{Wait}_B}(p_i)$ and $T_{\text{Wait}_T}(p_i)$, where $T_{\text{Wait}_B}(p_i)$ represents the time processor $p_i$ is awaiting for tokens before they are transmitted and $T_{\text{Wait}_T}(p_i)$ represents the time processor $p_i$ is awaiting for tokens while they are in *transit*.

The total idle time on $p_i$ is thus

$$T_{\text{Idle}}(p_i) = T_{\text{Fin}}(p_i) + T_{\text{Wait}_B}(p_i) + T_{\text{Wait}_T}(p_i)$$

(2.3)

From this discussion, we see that the use of characteristic timing parameters extracted from the underlying architecture, enables us to model the behavior of each individual processor and thus the entire parallel architecture.
2.2 Several Ways to Model the Computations

Concerning the parallel computations, various types of models have been proposed [35]. The most simple one specifies the computations as modules which are executed sequentially on the processor on which they are assigned and no communication takes place between the individual modules. Each module is labeled with an integer representing its execution time. We note that since no communication among the modules takes place, there are no restrictions on the execution order. In other words, there exists a total absence of precedence relations among modules in this computational model. Although scheduling of such independent modules – by the means of minimizing the total makespan – is an interesting problem, it may not, with a sufficiently accuracy, model a program characterized by sub-algorithms among which there exists a set of precedence relations.

A more accurate model is therefore required. Basically, communicating modules – represented as nodes – are interconnected via a set of directed arcs which indicate precedence relations. We restrict the precedence relations to feed forward only, i.e., nodes and arcs together form the Directed Acyclic Precedence Graph (DAPG). Each node is assigned an integer label indicating the actual or estimated run time. However, no labels are assigned to the arcs. We refer to this computational model as Precedence with No Cost (PNC).

Generating schedules based on the PNC model, we may still not obtain solutions which are sufficiently accurate. The problem is, basically, that the PNC model do not capture IPC and this may eventually turn out to be a severe limitation. An extension which highlights the IPC overhead can be easily constructed by labeling all arcs with integers corresponding to the total amount of tokens transferred between the nodes. Therefore, in this model which we denote Precedence With Cost (PWC), the DAPG holds information about 1) the expected node execution time and 2) the amount of communication between the nodes. The reason for annotating the DAPG with communication loads is to give the system designer the opportunity to experiment with various kind of interprocessor communication facilities like e.g., shared memory (SM) and message parsing systems which may not provide equal communication times for the same amount of transmitted tokens.

2.3 Formal Scheduling Notations

In chapter 3 we will launch our discussion about various static scheduling strategies but first we present our formal scheduling formulation upon which the strategies are going to be designed.

The overall problem is to schedule a DAPG onto a set \( S_p \) of \( P \) identical processors \( p_i \). Initially, we assume the PNC computational model, and thus we denote the DAPG as \( G = (S_n, S_a) \) where \( S_n \) is the set of nodes \( n_i \) and \( S_a \) represents the precedence relations among these nodes. For the graph \( G \), we define a function \( f \) which returns the estimated run time of a particular node, i.e., \( w_a = f(n_a) \). We formulate scheduling as the set \( \mathcal{F} \) of possible scheduling functions \( g \) which are able to map \( G \) onto a given target architecture \( S_p \). Now, \( g(n_a) \) returns the set of processors on which \( n_a \) is scheduled. Initially, when using the PNC model, \( n_a \) is scheduled on only one processor but in general we allow recalculation, and in such cases \( g(n_a) \) may return multiple processors.

Next, \( \forall g \in \mathcal{F} \exists g(p_i) \) is the set of nodes mapped onto processor \( p_i \) by the function \( g \). Similarly, \( \forall g \in \mathcal{F} \exists S_g \) where \( S_g \) denotes the set of valid schedules \( s \) which can be obtained using \( g \). Therefore, \( S_g \subseteq S_{\text{total}} \) where \( S_{\text{total}} \) is the set of all possible valid mappings of \( G \) onto \( S_p \), i.e., the solution space. For a schedule \( s \in S_g \), \( s(n_a) \) represents the actual start time of node \( n_a \). On the basis of these definitions, we are now able to validate a schedule \( s \). For \( s \) to be valid,
we require that the following conditions hold for any pair of nodes \((n_a, n_b) \in S_n\).

\[ \text{if } (n_a, n_b) \in S_a \text{ then } s(n_a) + f(n_a) \leq s(n_b) \]  
\(\forall p_i \in S_p, \forall n_a \in S_n, \ n_b \in \bar{g}(p_i) \)

\[ \text{if } n_a \neq n_b \wedge s(n_b) \geq s(n_a) \text{ then } s(n_b) \geq s(n_a) + f(n_a) \]  

The first condition states that the schedule must not violate any precedence relations specified by \(S_a\) and the second specifies that \(g\) must perform the mapping such that any processor executes only one node at a time.

Based on these definitions, we are now able to specify a set of equations which, for a given schedule \(s\), describes the various architectural states. First of all, the time the processors are using for calculations is

\[ T_{Calc}(p_i) = \sum_{n_a \in \bar{g}(p_i)} f(n_a) \]  

Next, since the computational model rely on “no communication cost”, \(T_{Comm}\) equals zero. Also \(T_{House}\) is zero because we assume 1) static scheduling 2) without recalculation, and 3) buffering of tokens is not required. Similarly, due to the absence of communication cost, the processors are never awaiting for tokens in transit, but the schedule has to obey the precedence constraints and thus a processor is likely to be forced into a \(T_{Wait}\) state. Subtracting the time, a processor \(p_i\) has spend calculating, from the time the last node has run to completion provides a measure for the time the processor spends waiting for nodes (on other processors) to complete, i.e.,

\[ T_{Wait}(p_i) = \max_{n_a \in \bar{g}(p_i)} \{s(n_a) + f(n_a)\} - T_{Calc}(p_i) \]  

Finally, the idle time associated to \(T_{Fin}\) on processor \(p_i\) equals

\[ T_{Fin}(p_i) = \max_{n_a \in G} \{s(n_a) + f(n_a)\} - \max_{n_a \in \bar{g}(p_i)} \{s(n_a) + f(n_a)\} \]  

where we note that the first term equals the makespan.

As we have indicated, generating a schedule on the basis of the PNC model may have some disadvantages — in particular when the communication times are comparable to or greater than the average node run time.

In order to overcome this drawback, the schedule generation must account for interprocessor communication time. We therefore initially have to discuss how accurate the communication cost model actually has to be. For the example, should we create schedules which in every respect take into account the tasks associated to communication (assigning time to e.g. \(T_{Init}\) and \(T_{Term}\)) or should we model only the time required for the physical transmission of a message, \(T_{Route}\)?

The answer to this question is closely related to the underlying architecture. In a non-fully connected message parsing multiprocessor architecture, for the example, \(T_{Route}\) may take up a substantial part of the overall communication time as compared to 1) a SM multiprocessor architecture (assuming all processors to have access to the same common data memory) or 2) a fully connected message passing architecture. In the case of the non-fully connected architecture, we therefore expect \(T_{Route}\) to be of more importance than \(T_{Init}\) and \(T_{Term}\) whereas for fully connected architectural types it may well be that only \(T_{Init}\) and \(T_{Term}\) has to be be represented in the communication model.
In any case, however, we have to rely on the PWC computational model. Strictly interpreting this model, we observe that it do not provide any information related directly to \(T_{Init}\) and \(T_{Term}\). The PWC model is attractive mainly in the sense that scheduling strategies can be devised which utilize the inherent IPC delay to make improvements to the makespan (recalculation may be an opportunity). In [35], a very simplistic interpretation of the PWC model is referred. We note that the authors unrealistically assume that all communication delays are equal to \(\tau\) — no matter the amount of tokens transferred between the DAPG nodes. Under this assumption, a formalized scheduling formulation may be described, but we instead propose a reformation which more realistically rely on the actual amount of tokens transmitted between two nodes — \(t_{IPC}(n_a, n_b)\).

In the PNC model, partial results are globally available upon termination of a node. In this model it was therefore sufficient to specify the start time of a node as \(s(n_a)\). Under the PWC model, zero intraprocessor communication is still assumed but zero IPC is not. Therefore, specification of the processor on which a node is being invoked is required, meaning that if \(n_a \notin g(p_i)\) then \(s(n_a, p_i)\) is undefined, otherwise it defines the start time of \(n_a\) on processor \(p_i\). Note that \(f\) do not need to be redefined because of our assumption of homogeneous architectures.

Now, in order for a schedule \(s\) under this model to be valid, the following conditions must hold for any two nodes \(n_a\) and \(n_b\) among which a precedence relation exists.

\[
\forall(n_a, n_b) \in S_a, \ \forall p_j \in g(n_b)
\]

\[
\text{if } p_j \in g(n_a) \text{ then } s(n_a, p_j) + f(n_a) \leq s(n_b, p_j) \tag{2.9}
\]

\[
\text{if } p_j \notin g(n_a) \text{ then } \exists p_i \in g(n_a) \mid s(n_a, p_i) + f(n_a) + t_{IPC}(n_a, n_b) \leq s(n_b, p_j) \tag{2.10}
\]

\[
\forall p_i \in S_p, \ \forall n_a \in S_n, \ n_b \in g(p_i)
\]

\[
\text{if } n_a \neq n_b \land s(n_b, p_i) \geq s(n_a, p_i) \text{ then } s(n_b, p_i) \geq s(n_a, p_i) + f(n_a) \tag{2.11}
\]

Here we assume that 1) \(i \neq j\), and 2) \(t_{IPC}(n_a, n_b)\) denotes the IPC time (incl. possible time for contention) between the nodes \(n_a\) and \(n_b\).

The first statement says that if \(n_a\) and \(n_b\) are scheduled on the same processor, then \(n_b\) cannot start execution until \(n_a\) has run to completion. If, on the other hand, \(n_a\) and \(n_b\) are scheduled on different processors, then \(n_b\) can only be invoked \(t_{IPC}(n_a, n_b)\) time units after \(n_a\) has terminated execution. Finally, all processors in the architecture can execute only one node at a time.

Due to possible recalculation, the total time the processors spend calculating is now defined as

\[
T_{Calc} = \sum_{n_a \in G} f(n_a) \tag{2.12}
\]

although one may argue that the time spend recalculating should also be counted in here since the processors are spending time on producing partial results — even though these calculations are redundant.

The recalculation time, however, is included in \(T_{House}\) as

\[
T_{Recalc} = \sum_{p_i \in S_p} \sum_{n_a \in g(p_i)} f(n_a) - T_{Calc} \tag{2.13}
\]

\(T_{Sched}\) and \(T_{Inter}\), which are also included in \(T_{House}\), equals in this model both zero since we assume 1) static scheduling and 2) that data is globally available after the \(t_{IPC}\) delay.

Due to this communication delay, however, it is relevant to distinguish between the waiting time for 1) tokens currently being transferred to the processor, and for 2) tokens which are not
yet transmitted when the processor becomes idle (waiting for precedence relations to be fulfilled). This is illustrated in figure 2.1.

![Figure 2.1: The time processor $p_1$ spends waiting before $n_c$ can be initiated — $Wait(n_c, p_1)$ — is divided into 1) the time while tokens required by $n_c$ are in transit, $Wait_T$, and 2) the time before they are transmitted, $Wait_B$.](image)

Now, for any node $n_j$ to be executed on $p_i$, its transit time is given as

$$Wait_T(n_j, p_i) = \min\{tau_{IPC}, Wait(n_j, p_i)\}$$

and the time before transmission of its input tokens is initiated as

$$Wait_B(n_j, p_i) = Wait(n_j, p_i) - Wait_T(n_j, p_i)$$

Using these terms, the total time processor $p_i$ spends waiting for 1) tokens in transit, and 2) tokens not yet transmitted are

$$T_{Wait_T(p_i)} = \sum_{n_j \in g(p_i)} Wait_T(n_j, p_i)$$

and

$$T_{Wait_B(p_i)} = \sum_{n_j \in g(p_i)} Wait_B(n_j, p_i)$$

respectively.

We finally note that $T_{Fin}(p_i)$ under the PWC model is equivalent to $T_{Fin}(p_i)$ derived for the PNC model in equation 2.8.

### 2.4 A SM IPC-Model with Improved Accuracy

From our discussion so far it is evident that researchers previously have formulated the scheduling problem from a mathematical point of view. It is also evident that these previous models for scheduling under the assumption of IPC are indeed unrealistic. Our purpose is therefore to devise more realistic IPC models which will be needed during design of the Multiprocessors Scheduling Pool.

We now assume that IPC in the underlying multi PDSP architecture takes place through a shared memory (and thus all processors are interconnected by the means of a shared bus). It is
a well-known fact that for a high number of processors, a shared bus interconnection strategy may provide bad performance due to the possible existence of severe bus contention. According to our initial specification of target architectures with only a few processors (see chapter 1), we therefore expect the SM architecture to be a reasonable alternative in selection of an appropriate multiprocessor architecture.

In this type of architecture, the communication mainly becomes a matter of executing memory access instructions. Therefore, in case two communicating nodes are assigned to different processors, the transfer mechanism is basically implemented as additional write and read instructions on the processors. This is illustrated in figure 2.2.

![Figure 2.2: A) The original two-node DAPG with a communication load equal to $c_{ab}$. B) The expanded graph with two additional communication nodes.](image)

Here a basic two-node DAPG is characterized by an IPC overhead proportional to $c_{ab}$ which (in case $n_a$ and $n_b$ are assigned to different processors) must be present in the schedule. According to the target architecture, we extend the graph with two additional nodes which represents writing and reading primitives — $W_a$ and $R_a$, respectively.

We aim for the model to be accurate, and thus we next have to discuss under which assumptions our expanded DAPG can be applied and how it should actually be scheduled. In the SM model (where communication is based on ordinary instructions performed by the processors), we realistically assume that computation and communication cannot take place simultaneously. This, in fact, is opposed to some authors (e.g., [36], and [37]) who rely on the (for SM architectures) unrealistic assumption of concurrent computation and communication. Next, the expanded DAPG has an inherent precedence relation between the write and the read node which, of course, must be respected. However, we do not require that the read instruction should follow immediately after the write instruction — we allow as much time to elapse between the two as required due to other nodes already scheduled. This is shown in figure 2.3.

We also realistically assume that the SM may be implemented using multi-ported random access memory. This allows for (at least) one simultaneous write and read instruction. The write-read precedence can thus be somewhat relaxed for those communications where $c_{ab} > 1$ as illustrated in figure 2.4. We define this scenario as overlapped communication. For all communications, though, we further require indivisible transmission, i.e., non-preemptive data transfer.

Although multiple accesses to the SM (in terms of simultaneous read/write) is allowed in our model, we do not permit multiple simultaneous writes or reads. Therefore, in case the memory is being written at the same time a node wants to store its partial result, this second memory access is realistically postponed as indicated in figure 2.5.

Based on these initial remarks we now discuss how the scheduling of computations and communication actually can be done according to our model. We initially assume that scheduling of the DAPG nodes is done by the means of any available scheduling function $g$ residing in our
Figure 2.3: Typically, it is impossible for the read instruction to execute immediately after the write node — in this case due to $n_c$.

Figure 2.4: Due to a multi-ported SM assumption, we allow one simultaneous read and write instruction. In this example, three words are transmitted from $n_a$ to $n_b$.

Figure 2.5: Simultaneous write (or read) is considered as being illegal in our model, and thus one or more memory accesses have to be postponed.

Multiprocessor Scheduling Pool.

Since these strategies basically 1) select a node, 2) assign the node to a processor, 3) determine the expected start time on that processor, and 4) finally delete the node from the DAPG, we are into a situation where we do not know whether the successor node(s) to the node just scheduled on processor $p_i$ will also be scheduled on that same processor. Due to this uncertainty we now have to make a decision. Assume a node $n_a$ with a successor $n_b$ (see e.g., figure 2.2) where $n_a$ has just been scheduled. It now seems as a bad strategy (in terms of wasted processor resources) to schedule the write operation ($W_a$) since it may perhaps not be needed if $n_b$ is
later scheduled on the same processor (remembering that zero intra-processor communication is assumed). Therefore, in general we decide not to schedule the “write node” immediately after its corresponding “production node”.

This decision has the following corollary:

- Every partial result stays until the end of the current sample period in the processor’s internal register file.

We expect this to be a fair assumption, especially for modern types of PDSPs with substantial on-chip memory capability.

As a consequence, once a partial result (produced by e.g. \( n_a \) which has already been scheduled) is required as input for a successor (e.g., \( n_b \) which is not scheduled yet), then a write operation first has to be scheduled on the processor that executed \( n_a \) initially. This is shown in figure 2.6.

![Figure 2.6: A partial result is written only to SM when a successor request it on another processor — in this example due to the scheduling order \{a, c, d, b\}.](image)

Another scenario likely to take place may be a node, which is already scheduled, and which has two or more successors (i.e., the node is a branch node) that operate on the same token(s). In such a case, the optimal solution is to schedule only one write operation (corresponding to the set of tokens to be transmitted to all successors), and this write operation should be scheduled just ahead of the first successor that requires the partial result. Other successors, which later require the same set of tokens, can then read the partial result directly from SM no matter on which processor they are to be scheduled. In order for this action to be valid, we need the following assumption.

- The shared memory is sufficiently large to store all partial results written to it within one sample period.

Similarly, if multiple successors to a node are to be scheduled on the same processor (but different from the processor on which the predecessor node is already scheduled) and they operate on the same set of tokens, then — according to our assumption of “data always available once in the processor register file” — only one read operation optimally has to be scheduled on the “successor processor”.

The primary idea with these assumptions and working definitions is to make our scheduling environment as realistic and optimal as possible (in terms of minimal makespan). However, in generating the schedule there are some practical circumstances which prohibit us from efficiently employing these assumptions. Basically, as discussed in appendix A, the DAPG is created directly from the SDFG. In the SDF notation, multiple successors, which operate on the same
set of tokens produced by a common predecessor node, receive these tokens from a fork node (see e.g., nodes 11 and 12 in figure A.6). Due to the expected zero runtime assigned for such fork nodes (in the corresponding DAPG), they are simply short-cutted before the DAPG is scheduled. It is therefore impossible (at least on the basis of the current version of our SDF2DAP converter) for the scheduler to know if the sets of tokens transmitted on different arcs from a branch node are equivalent. If they are, then our model is accurate but due to the uncertainty inherent in the DAPG, we have decided to schedule all write and read operations in order to generate realistic schedules. In doing so, we enforce an unnecessary overhead in those cases where the sets of tokens on the arcs from a branch node are the same. However, since we are aiming at an approach as realistic as possible, we prefer this penalty as compared to a reduced accuracy.

Now, for this computation/scheduling model we next investigate the various states in which the processors can operate. As usual, \( T_{Calc} \) equals the total sum of expected node run times, see equation 2.12. The time the processors spend housekeeping boils down to the recalculation time because of our static scheduling paradigm \( (T_{Sched} = 0) \) and since 1) we assume sufficiently large \( ij \) processor register file and \( ij \) SM and 2) buffering of tokens is performed simultaneously as they are written to the SM \( (T_{Inter} = 0) \). The recalculation time is defined as the total time spend computing minus the calculation time, see equation 2.13.

As compared to the other models which we have previously referred, our model accounts for communication time in a much more realistic fashion and thus \( T_{Comm} \neq 0 \). Due to the assumption of data exchange through SM, \( T_{Route} = 0 \), whereas \( T_{Init} \) and \( T_{Term} \) represent the actual communication overhead. The former reflects the time spend copying data out of the on-chip register file to the SM, i.e., write operations. Similarly, the latter equals the time required for copying data from the SM into the on-chip register file, i.e., read operations.

According to our argumentation above we found that for every DAPG arc which in the schedule imply an IPC transaction, a write as well as a read operation has to be implemented. From this we conclude that the total time spend in the overall architecture for transmissions to the SM equals the total time spend for transfers from the SM.

However, in general we may also conclude that \( T_{Init}(p_i) \neq T_{Term}(p_i) \). In order for us to determine the IPC times on the individual processors, we next redefine the function \( f \) so it may also returns the execution time associated to the communication nodes; \( f(n_{W_a}) \) denotes the time required to write a set of tokens from node \( n_a \) to SM and \( f(n_{R_b}) \) returns the time for reading a set of tokens to node \( n_b \) from SM. Employing this notation, the communication time on the individual processors is therefore

\[
T_{Comm}(p_i) = \sum_{n_{W_a} \in g(p_i)} f(n_{W_a}) + \sum_{n_{R_b} \in g(p_i)} f(n_{R_b}) \quad (2.18)
\]

where \( n_{W_a} \) and \( n_{R_b} \) represent nodes assigned to \( p_i \) for off-chip writing and reading, respectively.

In order to complete the description of various processor states, we finally go into a discussion of the idle time which was defined in equation 2.3. Initially, we concluded that \( T_{Fin}(p_i) \) is unchanged — i.e., the makespan subtracted the finishing time of the final node scheduled on \( p_i \) as indicated on figure 2.7. On the other hand, \( T_{Wait_B} \) and \( T_{Wait_T} \) need to be defined separately. Figure 2.7 shows an example with a precedence relation among \( n_a \) and \( n_b \). As usually, \( Wait(n_i, p_i) \) is divided into 1) the time before the token(s) are transmitted, and 2) the time while they are in transit — denoted by \( Wait_B(n_i, p_i) \) and \( Wait_T(n_i, p_i) \), respectively. However, since the partial result from \( n_a \) in our model is not globally available upon termination of the calculation, then it doesn’t make sense to talk about “tokens in transit” before the write op-

---

\(^1\)This node may be a communication node in terms of a “write”
eration is initiated. The time that elapses until that moment is defined as $Wait_B$. Now, in the previous unrealistic models, $Wait_T$ represented the "transit time" until the successor node starts execution (which is illustrated by the diagonal arrow). In our new model, this strategy cannot be maintained due to the receiving node $n_{R_b}$, which has already been counted into $T_{Comm}$ on the processor where the successor is scheduled. Therefore, although the token(s) are still in transit, $Wait_T$ do not include the time required for reading the SM. We realistically assume the "receiving node" to be scheduled just ahead of the "successor" node which means that no gap never exist between the two and thus a further contribution to $Wait_T$ is not relevant – at least for this particular situation.

Figure 2.7: A typical schedule with indications of the three different kinds of idle times.

There may, however, exist another situation where $Wait_T$ should be extended beyond the time the transmitting processor spend writing the partial result into SM. There is no guarantee what so ever that corresponding writes and reads are scheduled immediately after one another. This is basically due to 1) imbalanced load distribution and 2) contention for communication resources. An example is given in figure 2.8. Here, the DAPG is assumed scheduled according to the order $\{a, b, c, d, e\}$. The letters ($h$) and ($l$) on the graph arcs indicate high and low communication overhead, respectively. The number of processors is raised to three (due to the fact that for two processors only, prolonging $Wait_T$ because of communication contention will never occur\(^2\)). With more than two processors, a communication already scheduled can occur in between the communication operations currently being scheduled — like $n_{R_a}$ in figure 2.8. In such a situation, the transit time is prolonged, as indicated by the asterisk.

It may also turn out, that the idle time is 1) pure transit time, or 2) there may be no idle time at all. Two such scenarios are indicated on figure 2.9.

Based on these observations we conclude the following for our model. For any node $n_j$ to be executed on $p_k$ and with a predecessor on $p_l$ where $k \neq l$, the transit time is given as

$$Wait_T(n_j, p_k) = \min\{[s(n_{R_j}, p_k) - s(n_{W_j}, p_l)], Wait(n_j, p_k)\}$$

where $s(n_a, p_l)$ denotes the start time of $n_a$ on $p_l$.

Similarly, the waiting time associated to precedences before the set of tokens is transmitted is in our model given as

\(^2\)The reason being that at the moment communication is scheduled, the write node is always the last node on the transmitting processor and thus the corresponding read node (scheduled on the receiving processor after but not necessarily immediately after termination of the write node) cannot conflict with another read node on the transmitting processor.
CHAPTER 2. SCHEDULING AND MODELS

Figure 2.8: In this example, the read operation for $n_d$ is postponed due to current occupation of the communication channel. As a consequence, the transit time for tokens to $n_d$ is prolonged similarly.

Figure 2.9: A) The idle time on processor $p_1$ is pure transit time. B) Although communication is scheduled onto $p_1$, it has no idle time.
\[ \text{Wait}_B(n_j, p_i) = \text{Wait}(n_j, p_i) - \text{Wait}_T(n_j, p_i) \] (2.20)

Since these values represent the individual \textit{transit} and \textit{before} times on the various processors, the total time associated to these types of idle times can be found using equation 2.16 and 2.17.

### 2.5 An IPC Model for PDSP Message Passing

The shared data bus in a SM architecture is known to be a potential bottleneck if 1) the number of processors is raised substantially or 2) the graph granularity becomes too small. Therefore, a number of PDSPs (e.g. [21] and [23]) has introduced IPC facilities in terms of on-chip dedicated communication hardware which make \textit{simultaneous communications} among multiple processors possible. In order to facilitate scheduling onto such types of architectures as well, we next discuss our model for a multi PDSP system which rely on \textit{message passing communication channels}.

Since we are limiting ourselves to non-massively parallel target architectures, it is realistic to assume architectures that are fully connected, i.e., \textit{all processors are able to intercommunicate simultaneously}. Furthermore, state-of-the-art PDSPs with dedicated communication links also typically incorporate a CPU-independent on-chip DMA-coprocessor which alleviate the problem of CPU overheads required for communication related data transfers, e.g., transfer of tokens between on-chip memory and the dedicated communication hardware. The primary consequence of having an on-chip DMA-coprocessor is that \textit{communication and computation can be overlapped}.

Therefore, if we, according to figure 2.2, launch our discussion concerning an appropriate IPC-model, we realize that there is no need to expand the DAPG with additional communication nodes when scheduling onto message-passing PDSP architectures — the write and read nodes required are assumed to be executed by the DMA-coprocessor \textit{simultaneously} with numerical computations performed by the CPUs. An example is shown in figure 2.10.

![DAPG diagram](image)

Figure 2.10: A four-node DAPG is scheduled onto a two-processor message passing architecture. Note that \( p_0 \) is busy doing numerical computations while the communication (having a time complexity proportional to \( c_{ab} \)) takes place.

We next realistically assume that the dedicated on-chip communication links are \textit{bidirectional half duplex} channels which, due to the severe \textit{pin limitations} on most PDSPs, are transmitting
a word in several tempos\(^3\). Refering to figure 2.10, we now discuss the various operations which are included in our architectural model.

The assumption made previously for SM architectures concerning partial results which are automatically written to and stored in an on-chip register file is also employed for the message passing architecture. Basically, it means that when a node terminates, the partial result produced is available from the processor until the end of the sample period. Now, in case the partial result is requested by a node which is assigned to another processor, the DMA-coprocessor will be activated. Primarily, the DMA-coprocessor

- fetches the actual set of tokens from the on-chip memory
- writes the set of tokens to the output register associated to the appropriate communication port

Writing a data word to a communication port register typically initiates the data transmission on the channel. However, due to the assumption of half duplex capability, the transmitting processor (say \(p_0\)) do not have immediate link access if the processor to which it is connected (say \(p_1\)) is currently master of the link — processor \(p_1\) is for example transmitting on the link at the moment when processor \(p_0\) request the communication hardware, as illustrated in figure 2.11. We therefore introduce a **set up time** in order to model the time required to 1) become link master and/or 2) wait for possible contention to be solved.

![Diagram](image)

Figure 2.11: Since half duplex capability is realistically assumed, data transmissions may be postponed due to occupied communication hardware.

We note that state-of-the-art PDSPs with dedicated communication links have on-chip hardware facilities to handle the link arbitration problem, i.e., hardware which according to a handshake protocol can assign mastership to the processor which needs the link for data transmission. Basically, this can be done as follows, [21].

- Processor \(A\), which do not own the link, requests ownership.

\(^3\)A 32-bit floating point word, for the example, may be transmitted as eight 4-bit words.
• Processor $B$, which currently owns the link, acknowledge the request (possibly after completing a transmission) and relinquish ownership to the requesting processor.

• A “token” is thus transferred from one hardware control unit to another which means that the link ownership has changed from the one processor to the other and a data transfer from $A$ to $B$ can next be initiated.

Therefore, since 1) the DMA coprocessor handles data transfers between local memory and the I/O port registers, and 2) hardware control units are responsible for link arbitration, we safely assume an interprocessor data transfer to take place at the same time as the CPU is active. We note that the CPU and the DMA controller may access a common resource simultaneously, e.g., the local on-chip memory. In order to take care of that situation, we make the solid assumption of available arbitration logic which assigns priority to one of the two, delaying access for the other. However, we reasonably assume that this scenario occurs only occasionally and therefore our model do not take this type of competition in account. Beside, the core processor is usually responsible for activating the DMA mechanism. We assume that this control overhead is negligible as compared to the computations being performed on the CPU, and therefore is abstracted out of our model.

![Diagram](image)

Figure 2.12: In order to transmit a set of tokens from $n_a$ to $n_b$ assigned onto different processors, 1) the on-chip DMA coprocessors, 2) link arbitration logic, and 3) the physical channel are employed.

Based in these assumptions, the transmission scenario is modeled as a sequence of actions, as illustrated in figure 2.12. Basically, the DMA mechanism (on the transmitting processor) transfers the set of tokens from the local memory to the output register (indicated by DMA R/W). When the first token arrives at the output buffer, the arbitration logic request the link. If occupied, the transmission is prolonged until the arbitration logic gains link ownership. The output buffer, of course, has a finite capacity which restricts the number of tokens that can actually be written to the buffer. We assume, however, that the output buffer (and the input buffer on the receiving processor as well) are sufficiently large to store all the tokens in any transmission. As a consequence, the DMA read/write time is made proportional to the number of tokens which are to be transmitted.

From a scheduling point of view, we note that this communication model has a very important distinction as compared to the shared memory communication model. Since in shared memory types of architectures the processors (CPUs) handle the communication (i.e., the IPC mechanism is based on instructions to be scheduled together with the computational instructions), we decided that an IPC transaction for shared memory architectures can only be scheduled after the latest node which (up to the current point in time) is assigned to the transmitting processor. The reason is that we — as a first approach — enforce the communication scheduling and the compu-
Scheduling to take place simultaneously. Due to the fact that dedicated communication hardware is present in PDSP message passing architectures, an IPC transaction in these type of architectures do not interfere with the computational instructions and may thus be scheduled immediately after the node which has produced the required partial result (assuming, of course, that the IPC hardware is available).

As noted previously, the communication channel may be occupied. This implies that the communication hardware will be busy waiting until the communication resource becomes idle. We assume that within this waiting time (no matter how short it may be), the arbitration logic is able to gain link ownership for the transmitting processor. It means that the setup time, as illustrated by figure 2.12, models the time which elapses from the link is requested and to the “receiving processor” finalizes its current transmission. The dashed line after $n_a$ on figure 2.14 indicates an example of a setup time.

On top of this discussion, we now briefly describe the various processor states as defined in section 2.1. The time spend calculating can be found employing equation 2.12. Similarly, the time required for housekeeping can be found using equation 2.13 (recalculation only due to 1) the static scheduling scheme ($T_{Sched} = 0$) and 2) the DMA-coprocessor which handles required buffering ($T_{Inter} = 0$). Next, the presence of dedicated communication hardware has the essential impact that we assume $T_{Comm} = 0$ for message passing DSP systems. Beside, the idle time must be calculated in a slightly different way. First we note that $T_{Fin}$ is unchanged as compared to the previous discussion. Therefore, we concentrate on $T_{Wait_B}$ and $T_{Wait_T}$. Figure 2.13 shows the basic definition of “message passing waiting time”.

![Diagram](image)

Figure 2.13: Overlapped computation and communication is allowed due to dedicated communication hardware. The time a processor is awaiting for a message (i.e., a set of tokens) is still divided into 1) the time before and 2) while the message is in transit.

We note that $T_{Wait_T}$ corresponds to the time required for all communication operations (as depicted in figure 2.12), and $T_{Wait_B}$ denotes the time before the message transmission is initiated. This scenario do not suffice when it comes to formal definition of the two types of waiting times. Basically, contention for the link between two processors is an important behavior which we have to take into account. The basic mechanism is outlined on figure 2.14.

In this example, we specify the scheduling order to be $\{a, c, d, e, b\}$ and thus $n_a$ requests

---

$^4$ As we will see later, this may not always be the case since communication may also be scheduled after the DAPG nodes are assigned to and ordered on the various target processors.

$^5$ We note that there may be a CPU overhead in terms of e.g., register initializations, associated to the communication. This, however, is not included in our model.
Figure 2.14: Contention for the interconnection link occurs when $n_b$ requests the partial result from $n_a$ while there are ongoing traffic. This is indicated by the solid arrow from $n_c$.

the partial result from $n_c$ before $n_b$ requests data from $n_a$. Note that according to the actual communication time (from $n_c$ to $n_b$), the set of tokens is available before (and is being kept on $p_0$) until the initialization of $n_c$. Therefore, when $n_b$ request the partial result from $n_a$, the scheduler should launch this communication immediately after $n_a$ has terminated. The communication channel, however, is occupied and $p_1$ has ownership of the link. It is likely, that the DMA mechanism immediately can initiate its data transfer (from memory to output register) of the result from $n_a$. This, however, may give rise to DMA contentions which we do not include in our model. Although we are likely to introduce an over-estimated communication time, we safely postpone the complete communication until the ongoing communication has terminated — shown in the figure by the combination of a dashed and a solid diagonal arrow. From this, we now conclude that 1) the time $T_{Wait_B}$ may also correspond to waiting time imposed by link competition, and 2) $T_{Wait_T}$ denotes the actual transmission time.

Figure 2.15: A scenario indicating how the waiting time is dedicated to respective nodes.

We emphasize that the waiting time $Wait(n_i, p_j)$ corresponds to the idle time right in front of node $n_i$ on processor $p_j$. Figure 2.15 shows an example where it might be easy to mix up different waiting times. Although the set of tokens transmitted for $n_c$ is also in transit during
CHAPTER 2. SCHEDULING AND MODELS

the time denoted $Wait_T(n_b, p_i)$, we define this time period as being waiting time associated to $n_b$. It is obvious that a specific time slot on any processor can be associated to only one type of operation — otherwise it may be counted in multiple times.

Upon these remarks we finally present our equations for waiting time calculations. We define the time associated to tokens in transit as

$$Wait_T(n_j, p_k) = \min\{s(n_j, p_k) - link(n_j, p_l), Wait(n_j, p_k)\} \quad (2.21)$$

where $k \neq l$, and $link(n_j, p_l)$ denotes the starting time of the data transmission from processor $p_l$ to the processor on which $n_j$ is scheduled.

The time spend waiting before the set of tokens is brought into transit is defined as the difference between the overall waiting time and the transit waiting time.

$$Wait_B(n_j, p_k) = Wait(n_j, p_k) - Wait_T(n_j, p_k) \quad (2.22)$$

As an additional example, figure 2.15 is used to illustrate a Scheduling Order Rule which we will employ in those of our scheduling strategies which successively selects a node and assign it to a processor;

- Once a node $n_i$ is scheduled, no other nodes $n_j$ will never be scheduled ahead of $n_i$ on the same processor although 1) there might be room enough and 2) no precedence constraints will be violated.

If, in the example, the partial scheduling order was \{c, e, b\}, $n_b$ would then reside after $n_e$, although $n_b$ obviously, according to figure 2.15, is potential for being scheduled ahead of $n_e$. Note that the communication from $n_a$ to $n_b$ may still be scheduled as shown in the figure. The reason why we are employing this rule is that we want to keep the computational complexity of the scheduling heuristics as low as possible although the rule may prevent the optimal schedule from being found.

2.6 A Note on Synchronization Mechanisms

It is well-known that in parallel processing the individual processors must synchronize in order to exchange partial results so that the precedence relations among the individual subalgorithms (or nodes) are fully maintained. We do not intend to go into much detail about this subject here, but instead we refer to [38] which gives an exhaustive survey of existing synchronization mechanisms for MIMD types of machines — which is also the class of multiprocessor architectures to which our target machines belong.

However, to finish up this chapter, we outline the assumptions for synchronization principles on which we rely when it comes to scheduling. Basically, we divide synchronization into two classes according to shared memory and message passing architectures, respectively. Initially, we note that for fully static scheduling, application of a synchronization mechanism is irrelevant, simply due to the fact that the start times for computations as well as for communications are known exactly at compile time. This means that the processors are being synchronized at the time we generate the multiprocessor program. In this work, however, we rely on estimates for the run- and IPC times and thus variations as compared to the actual run and communications times are to be expected. According to this circumstance, synchronization among the processors is required.
For PDSP target architectures based on a shared data memory which is accessed through a shared interconnection medium in terms of e.g., a bus, priority is typically assigned to the individual processors in order to solve bus contention. Normally, a centralized controller is in charge of granting the bus to the processors. However, since in static scheduling (with estimated times), the start times for computations and communications, respectively, are "almost" known at compile time, we realistically assume the existence of a controller which provides access to the shared memory in the predetermined order. The processors request bus access by signaling the controller, and enter next idle mode (an operation normally provided by state-of-the-art PDSPs) if the bus is currently operated by another processor. According to this scenario, the synchronization is maintained by external hardware facilities, and only a very limited software overhead is needed on the individual processors for 1) controller request, and 2) processor idling. We therefore realistically ignore this minor synchronization overhead when generating our schedules.

For target architectures based on PDSPs interconnected through dedicated communication links, we realistically assume the same small amount of synchronization overhead as described above for SM architectures. For these types of architectures, the on-chip DMA coprocessor transfers data between the output/input register associated to every link. Synchronization is automatically maintained since a DMA-read, for the example, from an empty input register file (i.e., the set of tokens required for a node hasn't arrived yet) typically generates a not ready signal which may halt the CPU and let the independent DMA mechanism repeat its reading of the input register file until data becomes available.
Chapter 3

List Scheduling Heuristics

In chapter 2 we gave an introduction to our architectural and computational definitions which are required for the design of strategies for static scheduling. We did also mention that researchers previously have looked into the many interesting questions which naturally arise when a combinatorial optimization problem like scheduling has to be solved. We argued that previous works in the field of static multiprocessors scheduling have suggested some techniques which, however, may not provide realistic and efficient solutions. The purpose of the present chapter is therefore to explain, compare, and evaluate some of our early efforts devoted to improvements of existing techniques.

However, we are aware of the fact that research within the domain of scheduling is not as straightforward as it initially may seem. This is heavily influenced by a phrase from [39];

"The general scheduling problem is a fascinating challenge. Although it is easy to state, and to visualize what is required, it is extremely difficult to make any progress whatever toward a solution. Many proficient people have considered the problem, and all of them have come away essentially empty-handed. Since this frustration is not reported in the literature, the problem continues to attract investigators, who cannot believe that a problem so simply structured can be so difficult — until they have tried it!"

Despite that we know the existence of this frustration, we have launched our investigations without any fears of arriving empty-handed...

3.1 The Scheduling Technique Proposed by Hu

Among the first algorithm proposed for task scheduling onto multiple machines is The Hu Algorithm, [32]. Hu stated the problem as follows,

- "N jobs with ordering restrictions (i.e., precedence relations using our terms) have to be done by machines of equal ability (i.e., on a homogeneous architecture)".

1. Find the schedule which requires the minimum number of machines, so that all jobs are completed within a prescribed time $T$.

2. Assume $P$ machines — find a schedule which completes all jobs as soon as possible.

Although the former problem is of importance to the DSP community, where $T$ is identified as the sample period and typically is a critical parameter, we limit ourselves to studies of the
second problem.

Searching for a solution, Hu actually made some simplifications in that he assumed 1) equal runtime for all jobs (i.e., \( w = 1 \)) and 2) immediate "take over" for all machines — i.e., no setup-time when a processor finishes a job and the successive job starts executing on another processor.

Further, Hu assumed that the ordering restrictions among jobs form a tree — i.e., the problem graph has one and only one final node. The reason being that Hu considered scheduling as an assembly line problem where the final product is composed of several elementary parts. This, of course, put some topologically constrains on \( G \). Under these assumptions Hu derived a scheduling algorithm which maps a graph \( G \) onto a set of parallel machines, \( S_p \). The algorithm basically has two parts.

**Labelling:** Preprocessing \( G \) by specifying the node selection order.

**Node allocation:** Select nodes from \( G \) and assign them to available processors.

The labeling procedure basically assigns a value to every node indicating the critical path from that particular node to the final node. These values are next applied in order to select nodes having the highest need (or priority) for being assigned to a processor. The labeling procedure is given as

\[
\alpha_i = x_i + 1
\]

(3.1)

where \( \alpha_i \) is the label of node \( n_i \) and \( x_i \) is the length of the longest path from \( n_i \) to the final node in \( G \) which initially is given the label 1. The procedure is initiated from the final node and is then tracing backwards.

After the labeling procedure, nodes which have no predecessors are considered as starting nodes, i.e., nodes which have all input tokens available and therefore are ready for invocation. The basic strategy, which we outline in algorithm 1, then selects one or more starting nodes and assign them to the processors.

| STEP 1: If the total number of starting nodes is less than or equal to \( P \), then all starting nodes are available for allocation to different processors; |
| STEP 2: If the total number of starting nodes is greater than \( P \), choose \( P \) starting nodes with values of \( \alpha_i \) not less than those not chosen. In the case of a tie, the choice is arbitrary; |
| STEP 3: Remove the selected nodes from \( G \). This will initiate another set of starting nodes; |
| STEP 4: Repeat until all nodes have been assigned to a processor; |

**Algorithm 1**: Four steps are included in the plain Hu algorithm.

We illustrate the heuristic by the example shown in figure 3.1. Here, eight nodes, all of them with runtime equal to 1, are connected in a tree-like graph topology. Applying the labeling procedure, each node is assigned a value (the underscored number) corresponding to its location in the critical path.

Next, we assume that the graph has to be scheduled onto a two-processor target architecture. In figure 3.1 this is illustrated by the Gantt chart residing below the DAPG. According to the

\(^1\)Hu also showed that his algorithm finds the optimal schedule in terms of minimal makespan when 1) \( G \) is a tree and 2) all nodes have equal execution time.
Figure 3.1: Using the Hu algorithm for scheduling an eight node graph onto a two-processor target architecture. The emphasized numbers denote the order in which selected nodes are assigned to the processors.

Hu algorithm, the two starting nodes with the highest label are initially selected and assigned to the processors. In the graph, however, there are three starting nodes having identical labels. We thus arbitrarily select node 1 and 2. They are then removed from the graph which means that nodes 3 and 4 now become the starting nodes with the highest labels and are next assigned to the processors. This continues until all nodes have been scheduled. Note that in this heuristic, the assignment of a selected node to a processor is completely arbitrary, i.e., when a tie occurs, it is simply solved by arbitrary selection.

Now, in terms of scheduling an algorithm on a multi-PDSP architecture, the Hu heuristic may have some severe limitations. First of all, the strategy do not take into account the IPC time since it relies on the unrealistic assumption of immediate “take over”. Secondly, the heuristic will produce a load balanced solution which, in the presence of IPC, may not optimal.

According to our discussions in chapter 2 it is clear that the plain Hu heuristic is unable to produce realistic solutions unless the IPC can be ignored. In the following, we therefore propose new Hu-based heuristics which are more realistic.

3.2 An Improved Hu Algorithm

In this section we describe our proposals for a modification of the Hu algorithm. As indicated, the primary reason for making these changes is due to the unrealistic assumptions of

- equal node runtime
- immediate “take over”
- arbitrary node selection and processor assignment
• restricted tree graph topology

Concerning the node runtimes, we realistically allow all nodes in the (generalized) DAP graph to take on any value. With this modification we are no longer guaranteed to find the optimal solution in terms of minimal makespan\(^2\).

The labeling procedure remains unchanged but now we introduce a formal description in order to provide a basis for automation. We denote the DAPG, \(G = (S_n, S_a)\), where \(S_n = \{n_i : i = 1, \ldots, N\}\) represents all nodes in the graph. Based on this, we define \(S_a\) as

\[
S_a \equiv \{(n_i, n_j) \in S_n \mid n_i \rightarrow n_j\}
\]  

(3.2)

which explicitly express the precedence relations among the nodes in \(G\).

Assigning a label to a node requires that all successors to that node have already been assigned a label. For this reason, we next define a set of successor nodes \(n_j\) to every node \(n_i\).

\[
S_i \equiv \{n_j \in G \mid n_i \rightarrow n_j\} \quad i = 1 \ldots N
\]  

(3.3)

Each node is assigned an estimated run time overhead, \(w_i\), and these values are employed in the label calculation. Our first approach towards a “realistic Hu algorithm” is shown in Algorithm 2.

---

### Algorithm 2: An algorithmic description of our first approach towards a realistic scheduling heuristic based on the principle devised by Hu.

As compared to the plain Hu algorithm, our modifications allow arbitrary node runtimes which is definitely needed when we want to schedule DSP algorithms. Furthermore, we introduce a

---

\(^2\)The problem turns from being a “polynomial time” to an NP-complete combinatorial optimization problem for which no optimal algorithm is presently known.
CHAPTER 3.  LIST SCHEDULING HEURISTICS

Priority List, which in each level holds one or more nodes characterized by having the same label value. Our algorithm therefore belong to the class of heuristics known as List Scheduling Techniques, [40], where nodes are selected for processor assignment from a priority list. Finally, and most important, using the priority levels, we select nodes one by one instead of selecting $P$ at a time.

Note that during selection among nodes residing in the same level, we give highest priority to nodes with the longest execution time. The argument for this strategy is related to the philosophy of critical path execution. Although in the same level — and thus having the same initial position in the critical path — nodes with long execution times should probably be executed first in order to continuously minimize the remaining critical part.

These modifications are only a first approach towards a highly improved Hu algorithm. So far we have not included mechanisms for IPC handling and therefore we now describe ideas for further refinements which take into account the interprocessor communication.

3.3 A Hu Inspired IPC Heuristic

When extensions for IPC handling have to be incorporated into our heuristic, it is essential to utilize information on the underlying target architecture characteristics. Since we do not aim at detailed crafting of the parallel DSP programs, we therefore need estimates for 1) node runtimes, 2) communication overhead as well as 3) informations on communication mechanisms as described in chapter 2.

In order to develop our heuristic, we initially rely on the architectural assumption that the processors are able to perform overlapped computations and communications. The questions concerning the actual interpretation in terms of shared memory and message passing architectures are discussed later.

The procedures described above for 1) calculating the labels and 2) generating the priority list remain unchanged. So do the strategy for selecting nodes from the priority levels. The IPC handling is incorporated by postponing the node start times by taking into account the time required to transfer the amount of tokens specified in the DAP graph between two processors. This IPC time is denoted $t_{ipc}$ in algorithm 3.

Note, that in this IPC-extended heuristic we still employ an arbitrary processor selection in case a node can be invoked at the same earliest time on multiple processors. Intuitively, this can have a negative influence on the overall solution (in terms of a longer makespan) since it may initiate an unnecessary occupation of the communication facilities — possibly delaying the access for other communicating nodes. This is explained graphically in figure 3.2.

Here a 3-node graph has to be scheduled onto a homogeneous 2-processor architecture. Assuming that nodes $n_A$, and $n_B$ have already been scheduled as shown. Since we realistically assume zero intra-processor communication time, $n_B$ can start execution immediately after $n_A$ and then $n_C$ can run right after $n_B$ on $p_1$. In this example, the runtime of $n_B$ equals the IPC time from $n_A$ to $n_C$ and thus $n_C$ can be initiated at exactly the same time on $p_2$. Assigning $n_C$ to $p_2$ will require more resources in terms of occupied communication hardware as compared to the situation where $n_C$ is scheduled onto $p_1$. In order to circumvent this problem, we therefore modify our heuristic so that a node which can be invoked on multiple processors at the same earliest time is assigned to the processor on which one or more immediate predecessor(s) are already scheduled.

In some cases, however, this strategy may also lead to a tie. In those situations, we select the processor which has stayed idle for the longest time, as shown in figure 3.3. The reason for this
# Select nodes from the priority list and assign them to processors:
level_counter = 0;
Do until all \( n_j \in G \) have been scheduled
level_counter + = 1;
Do until level level_counter is empty
Let \( n_j \) be the node in level level_counter which has the longest execution time \( w_j \);
If more nodes have the same \( w_j \), \( n_j \) is chosen arbitrarily;
Delete \( n_j \) from level level_counter;
For all processor in the target architecture
Calculate the earliest start time of \( n_j \) constrained by the two requirements
1) \( \forall (n_i, n_j) \in S_a, n_i \) must run to completion before \( n_j \) is invoked;
2) \( \forall (n_i, n_j) \in S_a, \) if \( n_i \) is assigned to processor \( p_k \) then \( n_j \) can at earliest be
invoked on processor \( p_k \) \textit{t}_{\text{ipc}} \) time units after \( n_i \) has run to completion;
If \( (n_i, n_j) \notin S_a, n_j \) is invoked as early as possible on any available processor;
If \( n_j \) can be invoked on various processors at the same earliest time then
the processor choice is arbitrary;
End_For;
End_Do;
End_Do;

\textbf{Algorithm 3:} In this heuristic, the starting time is made dependent on the communication
overhead required if predecessor(s) are previously scheduled onto other processor(s).

![Diagram of a directed graph and Gantt chart]

Figure 3.2: Due to the communication time \( b \), node \( C \) can be invoked on \( p_1 \) and \( p_2 \) at the same
earliest time which is illustrated by \textit{two} \( C \)s in the Gantt chart.

heuristic is that since multiple processors after all have been brought into use and occupation
of communication facilities is inevitable, then we try to \textit{load balance} the processors as much as
possible. It is evident, that for architectures with more than two processors, a tie may still
occur. If that happens, we finally make the processor choice using arbitrary selection. We have
entitled this Hu inspired IPC concept \textit{"Hu*"}. The overall algorithmic description can be found
in algorithm 4.

\subsection*{3.3.1 Realistic Scheduling with the \textit{Hu*} Concept}

From algorithm 4, we see that the processor selection in \textit{Hu*} is guided by a calculation which
takes into account

- all individual predecessors to the selected node
- the actual IPC times from these predecessors
The earliest (IPC dependent) start time is the latest time (i.e., the worst case time) found among all the possible start times.

This basic concept, however, only describes the IPC overhead in terms of the estimated communication time. In the actual implementation of Hu*, we make a more realistic calculation. Basically, we make the start time detection dependent on the current state of the communication hardware.

From the appropriate DAPG arc, we first detect the amount of data tokens which has to be transmitted. Next, for a shared memory architecture, we include the estimated "one token no-contention" transmission time (see figure 2.2) from which we calculate the total estimated communication time. We realistically assume that the time required for WRITE (to the shared memory) equals the time required for READ. Therefore, the estimated total communication time is simply equally divided into $t_{\text{write}}$ and $t_{\text{read}}$.

As an example, assume 1) $(n_i, n_j) \in S_m$, 2) $n_i$ already resides on processor $l$, 3) $n_a, n_b$, and $n_c$ are independent nodes scheduled as shown on figure 3.4, and 4) the $n_j$ start time has to be calculated from processor $k$. Remember that we rely on the assumption that all intermediate results reside (forever) in an internal memory on the processor on which they are originally produced. Therefore, $W_i$ is schedule only when it is needed, i.e., just before $n_j$ is to be scheduled. The question now arises — what is the earliest time $W_i$ can be initiated?

From our Scheduling Order Rule (which also applies to communicational instructions) it is evident that the earliest time for the write operation to be invoked is the time when the last node — scheduled so far on the actual processor (in this case $p_l$) — has run to completion. This is indicated as early-write.

Since we do not allow multiple simultaneous write operations we next detect the earliest time slot on processor $l$ which is large enough to hold the write operation without introducing write contention. In figure 3.4 we note that the time span between $n_b$ and the next write operation, $W_c$, is larger than $t_{\text{write}}$ and thus $W_i$ can be abut scheduled after $n_y$.

We recognize the realistic but conservative nature of this strategy — detecting a tiny overlap among two write operations may cause the schedulable write operation to be substantially delayed. This situation may eventually be avoided by minor rearrangements of the existing schedule. The drawback, however, is a considerable increase in the overall scheduling complexity, which we would like to avoid.
Algorithm 4: Pseudo code for the refined Hu based scheduling heuristic which takes into account the IPC overhead. We refer to this basic concept as the Hu* strategy.

To finalize our example, the corresponding read operation has to be scheduled on processor $k$. Similarly, the read can be invoked no earlier than $early\_write + t_{write}$ (unless we allow overlapped communication, figure 2.4) and not before the “final” node on processor $k$ has run to completion. Again, we must find the first “read-free” interval larger than or equal to $t_{read}$ starting at or later than $early\_read$.

The realistic implementation of the Hu* concept for message passing architectures relies basically on the same principles as just discussed for shared memory. We follow our model described in section 2.5. The start time calculations (and the succeeding schedulings) detect the first free gap on the communication link (after the predecessor) large enough to hold the actual
CHAPTER 3. LIST SCHEDULING HEURISTICS

communication.

3.3.2 Is the Processor Selection Procedure Optimal?

Before proceeding, we'll make some notes on our strategy developed for processor selection. In order to select a processor, Hu+ detects

- predecessors to the selected node,
- their stop times, and
- the earliest start time for the node to be scheduled

taking into account the IPC time required for transferring the necessary amount of data tokens.

Now, the detected earliest start time (used for processor selection) is calculated on the basis of the IPC time enforced by the predecessor which give rise to the worst case start time. This means, unfortunately, that in cases of multiple predecessors already scheduled on other processors (than the one selected), a later start time, as compared to the initially estimated one, must be expected. This, of course, is due to the fact that communication from these predecessor have to be scheduled and therefore may postpone the initially estimated start time for the selected node.

As a related consequence, it may turn out, that the actual start time (and thus the processor selected) is not the optimal one. We can easily explain the reason for this. Since we, during our search for the processor providing the earliest start time for the selected node, are not (in all cases) exhaustively evaluating the consequence of IPC from all predecessors, a better solution may exist. We make the bold conclusion though, that the increase in the scheduling complexity required for a more optimal processor selection may not compare favourable to the performance improvements to be gained.

3.4 Scheduling by the Means of Node Duplication

Since realistic scheduling must take into account the IPC times, it is evident that as few communications as possible have to take place. As indicated previously, one possible solution may be to cluster precedence related nodes on a single processor in order to perform sequential execution — of course at the price of minimized utilization of possibly inherent parallelism.

An alternative method proposed by Kruatrachue and Lewis, [37], suggests simultaneous solution for both requirements stated in the MIN/MAX problem.

- Exploitation of maximal parallelism
- Minimization of required IPC

The overall idea is illustrated in figure 3.5. The DAPG represents a situation where a node has multiple successors and thus the graph is characterized by a certain amount of inherent parallelism. We further assume that the communication times $C_b = C_c > 1$. Now, the trivial solution — the sequential one — is shown as Solution no. 1, and it consumes a total of 3 time units. Solution no. 2 indicates a 2-processor schedule with makespan equal to $2+C_b$. Obviously, a clustering of all nodes on one processor is a far better solution as compared to the parallel program when $C_b \gg 1$. But, due to the actual graph topology, it is possible to find a parallel solution with a makespan shorter than that provided by the sequential program — this is illustrated by
CHAPTER 3. LIST SCHEDULING HEURISTICS

Figure 3.5: From this figure it is seen that duplication may prevent IPC although the inherent parallelism is fully exploited.

*Solution no. 3.* Here, node $A$ and then node $C$ are assigned to processor $p_1$. Next, node $B$ can be assigned to processor $p_1$ or $p_2$ as was the case in Solution no. 1 or no. 2. Allowing node $A$ to re-run on $p_2$ as indicated by $A$, provides a situation where no IPC is required in order to invoke $B$ which means that the optimal solution is found.

The primary advantage of this technique is its ability to minimize 1) the makespan and 2) the required amount of IPC simultaneously for DAPGs with heavy demands for IPC as compared to the node run times—like the situation in figure 3.5. Obviously, the price for this facility is 1) the need for an increased program memory and 2) an overall degradation of processor utilization since partial results have to be calculated multiple times. The former problem can in most cases be ignored, and the latter is of no importance in our situation as long as we respect the initial specification requirements.

In [37], a very brief description of a heuristic based on multiple node invocations is given. This method is denoted *Duplication Scheduling Heuristic* (DSH). The explanation of the algorithm, however, is so unclear, weak and insufficient that we were not able to extract the ideas in order to redo the experiments. Consequently, we have devised our own version based on the general principle of node duplication. Our heuristic is entitled *Node Duplication Strategy* (NODUST).

### 3.4.1 The Working of NODUST/ASAP

NODUST is also a *list scheduling heuristic* but, as compared to $H_u^*$, the list (in the initial version) is not based on the critical path approach. Rather, we initially employ a level hierarchy using an *As Soon As Possible* (ASAP) technique, [41], leading to NODUST/ASAP. Basically, we do this in order to *expose the inherent parallelism* which means that the node selection is related to the *graph parallelism* and not to the *critical path*. Figure 3.6 shows an example DAPG and corresponding lists (or level hierarchies) generated using CP- and ASAP-techniques, respectively. The ASAP hierarchy is constructed according to the description in algorithm 5.

In algorithm 5 it should be noted that we define root nodes as nodes having no predecessors (which may contradict with some other literature where root nodes are identified as nodes having no successors).

Now, using the ASAP hierarchy, we next have to perform 1) node selection and 2) assignment onto the processors in the target architecture. The former is made identical to the selection procedure applied in $H_u^*$ (except, of course, that here we are selecting the nodes from the ASAP list) whereas the latter is somewhat more complex. We therefore separate the NODUST/ASAP
CHAPTER 3. LIST SCHEDULING HEURISTICS

Figure 3.6: An example DAPG and the corresponding CP and ASAP level hierarchies.

description into two different parts; 1) detection and 2) evaluation of nodes potential for duplication, respectively.

```
# Generate ASAP hierarchy:
ASAP_level_counter = 0;
Do until all nodes nj in DAPG have been assigned to a level
   ASAP_level_counter + 1;
   Determine the set S_root of root nodes currently in the DAPG;
   Assign all nodes in S_root to level level_counter;
   Delete all nodes in S_root from the DAPG;
End_Do;
```

Algorithm 5: The algorithm employed for generation of an ASAP hierarchy. Iteratively, the algorithm finds nodes without predecessors, i.e., root nodes, and assigns them to the current level.

Detection of nodes potential for duplication

This section explains algorithm 6 which presents our ideas for detection of nodes potential for duplication.

Referring to algorithm 6, it is seen that the NODUST/ASAP heuristic search through all ASAP levels, once at a time. For each node, nj, selected at the current level, we next have to solve two tasks; 1) find the earliest possible start time for nj on every individual processor pk (k = 1..P), and 2) detect the predecessors to nj which may advance its start time by duplication, i.e., detect potential duplication nodes.

Basically, our algorithm works as follows. We investigate start times for nj on pk without and with IPC taken into account, i.e., t_start(k) and t_start,IPC(k), respectively. Initially, both start times are set equal to t_ready(k) — the time when pk has finished its previous job and thus the earliest time it is available for initiating a new one.

Next, the algorithm locates predecessors to nj on other processors than pk, i.e., nodes which may prolong the start time due to IPC or, in other words, nodes which may advance the start time of nj if they are possibly duplicated. Based on these predecessors (if any), we iteratively search
# The algorithm for node duplication detection:

\[ \text{ASAP}_\text{level}_{\text{counter}} = \text{"top"}; \]

Do until all ASAP levels have been examined

Do until the current level is empty

Select the node \( n_j \) having the longest runtime;

In case of a tie, select arbitrarily;

Remove \( n_j \) from the current level;

For \( k = 1..P \)

Set \( t_{\text{start}}(k) \), the start time of \( n_j \) on \( p_k \), equal to the time when \( p_k \) is ready to invoke the next node, \( t_{\text{ready}}(k) \);

Set \( t_{\text{start,IPC}}(k) \), the start time of \( n_j \) on \( p_k \) taking IPC into account, equal to \( t_{\text{ready}}(k) \);

Locate predecessors to \( n_j \) on \( p_1 \neq p_k \);

If \( n_j \) has predecessor(s) \( n_t \) on \( p_t \) then

For all predecessors \( n_t \)

Find the predecessor finish time \( t_{\text{finish}} \);

If \( t_{\text{finish}} > t_{\text{start}}(k) \) then \( t_{\text{start}}(k) = t_{\text{finish}} \);

Find the temporary start time \( t_{\text{start,IPC,temp}}(k) \) for \( n_j \) on \( p_k \) taking into account IPC from the predecessor;

If \( t_{\text{start,IPC,temp}}(k) > t_{\text{start,IPC}}(k) \) then \( t_{\text{start,IPC}}(k) = t_{\text{start,IPC,temp}}(k) \);

If \( \{t_{\text{start,IPC}}(k) > t_{\text{start}}(k)\} \lor \{t_{\text{start,IPC}}(k) - t_{\text{ready}}(k) > w_1\} \) then

\( n_j \) is potential for duplication on \( p_k \);

End_For;

End_If;

Evaluate the predecessor(s) which are potential for duplication;

End_Do;

Point at next ASAP level;

End_Do;

\begin{algorithm}

\textbf{Algorithm 6}: An outline of our algorithm for detection of nodes potential for duplication.

\end{algorithm}

for the earliest start time with and without IPC. When all predecessors have been evaluated, \( t_{\text{start}}(k) \) and \( t_{\text{start,IPC}}(k) \) hold the earliest start time on \( p_k \). Now, one may ask what the start time \textit{without} IPC taken into account is actually good for. Our idea is to employ this start time as a parameter for detection of potential duplication nodes. It may turn out that the earliest start time on a given processor \( t_{\text{start}}(k) \) is later than the time when the required tokens are available \( t_{\text{start,IPC}}(k) \). In such a case, we consider duplication as being irrelevant, otherwise the predecessor may be potential for duplication.

When iterating through the predecessors, we determine whether the individual predecessors are potential for duplication on \( p_k \). Figure 3.7 illustrates a situation where a potential duplication node is detected. Assume this 3-node graph being part of a larger graph where some predecessors \( x \) and \( y \) have already been scheduled onto \( p_1 \) and \( p_2 \), respectively. Further, assume that \( n_1 \) and \( n_2 \) are next assigned to the processors as indicated. The timing in the overall schedule is of no importance here, so let’s assume that \( t = 0 \) equals the time when \( n_1 \) is invoked. Since \( n_3 \) now has to be scheduled, we first investigate its possible assignment on \( p_1 \). Here, when the IPC time from \( n_2 \) is assumed equal to zero, \( t_{\text{start}}(1) = 3 \), i.e., equal to \( t_{\text{ready}}(1) \). Similarly, IPC taken into account, \( t_{\text{start,IPC}}(1) = 5 \). Based on these informations, the algorithm next investigates the possibilities for duplicating \( n_2 \) onto \( p_1 \).

Since \( t_{\text{start,IPC}}(1) > t_{\text{start}}(1) \), the potential duplication status for \( n_2 \) is \textit{partly fulfilled} as indicated by algorithm 6. Our second requirement, \( t_{\text{start,IPC}}(1) - t_{\text{ready}}(1) = 2 > w_2 = 1 \) (which says that there is “room enough” for \( n_2 \) to be duplicated), is also meet and thus \( n_2 \) is a potential node for duplication. In figure 3.7 we indicate duplication of \( n_2 \) by \( 2^* \) and realize that the start time for \( n_3 \) can be advanced from \( t = 5 \) to \( t = 4 \) (using the fact that intra-processor
Figure 3.7: A 3-node DAPG (which is assumed part of a larger graph complex) has to be scheduled onto a two-processor target architecture. A duplication of node 2 onto processor 1 (denoted as $2^*$) will advance the start time of node 3 on processor 1.

communication time is realistically assumed equal to zero). According to algorithm 6, we also have to investigate the potential duplication of $n_1$ onto $p_2$. This, however, is left out here.

Evaluation of potential duplication nodes

From the discussion above, it is evident that at exit of the node duplication detection algorithm we may have multiple pairs of potential duplication nodes, and their corresponding processors. However, although re-invocation of these nodes can minimize the makespan — at least based on the knowledge obtained by this preliminary evaluation — we have to launch a more detailed investigation due to the fact that duplication has to be done without violating precedence relations and IPC times. We therefore next perform an analysis of all potential node/processor pair and then possibly select the most favourable one. Note that in this work we are considering 1) duplication of one predecessor ($n_j$) only, and 2) no duplication of predecessors to a potential duplication node. Surely, our algorithm may be extended by also taking into account these constrains but, of course, at the price of a highly increased computational complexity.

Figure 3.8: A specification of node names involved in finding the earliest possible start time for $n_j$. The boxes indicates that there may be more nodes carrying the names.

Prior to a detailed discussion of our algorithm for evaluation of potential duplication nodes, we specify the node notation as depicted in figure 3.8. As before, we assume that the node currently under investigation is $n_j$. Predecessors to $n_j$ which are detected as potential for duplication are henceforth denoted $n_t$ and predecessors to these potential duplication nodes are denoted $n_s$. Since the potential duplication nodes are related to a specific processor, we denote these node/processor pair ($t, k$). Using these definitions, we next present our algorithm for 1) evaluation, 2) selection, and 3) scheduling of potential duplication nodes, algorithm 7.
Algorithm 7: The algorithm devised for 1) evaluation of predecessor nodes detected as being potential for duplication, 2) duplication and 3) scheduling.

Initially, we use \( t_{\text{start IPC}}(k), k = 1..P \), in order to determine the processor \( (\text{temp}_\text{proc}) \) on which \( n_j \) can be invoked at the earliest possible time — constrained by the use of IPC but without duplication. Here we note that multiple processors may provide the same earliest time and in such a case our selection becomes random.

Basically, the algorithm searches through all ordered predecessor-processor pair \((t, k)\) detected previously. During detection of potential duplication nodes, our purpose was to find predecessors which possibly can advance the start time for \( n_j \) on \( p_k \) (section 3.4.1). In this detection procedure, we did not take into consideration the predecessors, \( n_t \), probably residing on \( p_k \) since the start time for \( n_j \) would not be influenced at all due to the assumption of zero intra-processor communication time.

When it comes to duplication, on the other hand, we do consider these predecessors. It may turn out that \( n_t \) has been previously scheduled on \( p_k \) — either as a regular scheduled node or as a duplication. If \( n_t \) already resides on \( p_k \) then the argument for duplication fails, and thus the pair \((t, k)\) is no longer considered as being potential for duplication. If \( n_t \) do not reside on \( p_k \) then it is still potential for duplication, and we next calculate the earliest invocation time on \( p_k \). This calculation takes into account all predecessors \( n_s \) scheduled on \( p_l \neq p_k \).

Now, if the start time of \( n_t \) is earlier than or equivalent to the time when \( p_k \) is ready to invoke the next node, then we have the most optimal situation for duplication — we define it as abutment duplication. It means that \( n_t \) can be invoked immediately after the previous node which has run to completion on \( p_k \). As an example, node 2* on figure 3.7 is scheduled abut to node 1.

It should be pointed out here, that in case abutment duplication cannot be performed, then we consider duplication as being out of question. One may argue that this is a very conservative strategy due to the fact that non-abutment duplication still could provide a start time for \( n_j \) which may be earlier as compared to the start time without duplication. However, as a first approach towards a NODUST/ASAP heuristic, we apply the optimal duplication strategy only.
Assuming now that abutment duplication for \( n_t \) is possible, we next have to schedule \( n_j \) on \( p_k \) — and we want to invoke it right after \( n_t \). This, of course, can only be done if the IPC requirements from predecessors (to \( n_j \)) on other processors are not violated. We have previously calculated the earliest start time for \( n_j \) on \( p_k \) (algorithm 6). However, in the meanwhile other predecessors to \( n_j \) may have been duplicated which provides a new situation. We therefore have to re-calculate the earliest start time of \( n_j \) on \( p_k \). In case the re-calculated start time succeeds the time when the duplicated \( n_t \) has run to completion, then the \( n_j \) start time is updated.

We now introduce the term abutment scheduling which indicates the situation where \( n_t \) is scheduled to execute immediately after a duplicated predecessor \( n_t \). Our algorithm checks for the possibility of abutment scheduling by testing if the earliest start time found for \( n_j \) precedes or is equal to the \( p_k \) ready-time plus the runtime associated with \( n_t \). If so, then \((t, k)\) is added to the list \( pot\_sch\_list \), otherwise it is not. Again, this may be regarded as a conservative strategy, but for these initial investigations we use this technique as it provides the earliest invocation of the node.

Finally, in case the \( pot\_sch\_list \) actually holds one or more \((t, k)\) pairs, these are next evaluated in order to find the pair which provides the earliest start time for \( n_j \). When found, \( n_t \) is duplicated on \( p_k \) and \( n_j \) is scheduled — both of these assignments performed according to the abutment rules, as defined above. If, on the other hand, \( pot\_sch\_list \) is empty, \( n_j \) is simply scheduled onto processor \( temp\_proc \), detected as described previously.

3.4.2 The Working of NODUST/CP

The fundamental principle applied in \( Hu^* \) relies on the fact that nodes are selected according to their appearance in the critical path. Intuitively this seems as an appealing approach because a node currently in the critical path requires an earlier invocation time as compared to nodes which are not in the critical path. Above, however, we argued that a level hierarchy generated according to an ASAP schedule may be advantageous in scope of the NODUST philosophy because we want to expose the inherent parallelism. In order to verify whether this is a reliable idea or not, we devised an alternative NODUST version, the NODUST/CP.

The only difference as compared to NODUST/ASAP is that the level hierarchy is generated according to the critical path using exactly the same method as employed by \( Hu^* \) (see algorithm 2). This influences only the order of the node selection.

3.4.3 The Application of NODUST

So far we have considered the detailed working of the NODUST heuristics. In order to employ our techniques on specific multiprocessor architectures, we next have to discuss in which environments they actually fit.

From the introduction to the NODUST philosophy it is evident that detection and evaluation of potential duplication nodes are done according to the assumption that computation and communication can take place simultaneously (see figure 3.5). It is therefore immediately obvious, that NODUST suits message passing architectures which safely respect this assumption.

The overall NODUST principle which simultaneously tries to utilize inherent parallelism and minimize communication may also apply to shared memory types of architectures where separate communication operations are scheduled along with the program instructions. We consider, however, the invocation times with/without duplication for such architectures to be somewhat complicated to calculate — basically due to the possible separation of write and read instructions. Tuning NODUST to fit properly into a shared memory environment will require an extensive algorithm reformulation which we have chosen not to investigate in this work.
CHAPTER 3. LIST SCHEDULING HEURISTICS

We therefore only consider NODUST (in its present form) to be operated in a message passing architecture that comply with our computational model described in section 2.5.

This decision, however, has a more global consequence. In order to make a fair comparison between the individual strategies, experiments have to be performed on the same architectural platform. In the following we therefore limit our experimentations to message passing architectures only.

Implementational limitations of the NODUST scheme

Similarly to the limitations discussed for the Hu* strategy concerning the calculation of earliest possible invocation time for the selected node $n_j$, NODUST has the same inherent weakness. The calculated start times are based on the IPC transaction which imply the worst case start time. In other words, we do not perform a prescheduling of the total IPC pattern for $n_j$. It may mean that when the processor for $n_j$ has been selected (based on the earliest start times provided by our calculations), this processor may not provide the earliest invocation time after all, due to the actual communication schedule which also has to be done — i.e., when all IPC transactions are scheduled without contention onto the communication facilities, another processor may be more attractive in terms of the earliest invocation time.

Furthermore, we note that the detection of potential duplication nodes is also based on the worst case IPC transaction and thus duplication may not be attractive when all communications are scheduled.

Finally, above we defined the terms abutment duplication and abutment scheduling. We have designed our algorithm in the very conservative manner where duplication is performed only when we obey this abutment rule. In the implementation of NODUST, however, duplication may still be performed although the abutment rule is possibly violated due to exhaustive communication scheduling. The reason being that we choose duplication on the basis of the worst case IPC transaction.

3.5 Comparing Hu* and NODUST Using Random Graphs

Now, in order to give some indications towards the performance of the proposed algorithms, we next make a number of experiments based on randomly generated test graphs. Although these graphs are derived by a stochastic procedure there are, however, a number of parameters which are set at specific values in order to control essential properties of the graphs. Appendix B details our Graph Generator Tool (GGT) which is employed to produce the random graphs. Initially, we base our investigations on graphs characterized by different:

- number of graph nodes
- inherent parallelism

3.5.1 Graph Topologies

Basically, we define nine different graph sets as characterized in table 3.1 (see also section B.4). Due to the stochastic nature of GGT, each set contains five graphs (having, of course, identical values for the various graph properties) which are scheduled separately. The resulting speedup performance is finally calculated as the average of the five individual speedup measures returned from each set. One may argue that every individual graph should be scheduled a number of times due to possible stochastic selection applied in the scheduling heuristics (in case of a tie,
Table 3.1: Nine sets each containing five DAP graphs constitute our library of randomly test graphs. This table shows for each set 1) the number of nodes in the graphs, 2) the average degree of graph parallelism, 3) the average ratio between total communication and total computation, and 4) the average ratio between mean communication and mean computation.

Node/processor selection may be done randomly). Because of the extensive time requirement, however, we avoid multiple scheduling of the individual graphs.

The chosen interval of parallelism is controlled by varying the depth and the width of the graph. Additionally, 1) the node weights and 2) the number of tokens transmitted on the individual arcs are limited to certain intervals. These intervals are selected such that the average computation time is \( \mu_{\text{comp}} = 12 \) and the average number of communicated tokens is \( \mu_{\text{comm}} = 3 \) (assured by using a uniform distribution for random weight- and token selection). For all graphs, the expected value of the ratio \( \mu_{\text{comm}}/\mu_{\text{comp}} \) is therefore \( \frac{1}{4} \) — which from table 3.1 also can be seen to hold true.

Further, we calculate for each set the average of the ratios \( \rho \equiv \frac{\sum_{\text{comm}}}{\sum_{\text{comp}}} \) (where \( \text{comm} \) and \( \text{comp} \) represent \( c_{ij} \) and \( w_{ij} \), respectively, and the summations are over all nodes and all arcs). We now argue that these \( \rho \)-values reflect the interconnection topology of our test graphs. If we assume a GGT generated test graph where

- all nodes (except the leaf node(s)) have one and only one successor, and
- the number of leaf nodes \(<< N\)

then the number of precedence links \(~ \sim N\).

For such a graph one may say that the interconnection topology is balanced in the sense that the number of precedence links into a node near the root of the graph (on the average) is comparable to the number of precedence links into a node near the bottom of the graph (see also appendix B).

Since we define the average communication and average computation for a graph as

\[
\mu_{\text{comm}} = \frac{\sum_{\text{comm}}}{\#\text{links}}
\]

(respectively)

\[
\mu_{\text{comp}} = \frac{\sum_{\text{comp}}}{\#\text{nodes}}
\]

it then follows that
\[ \rho = \frac{\# \text{links}}{\# \text{nodes}} \cdot \frac{\mu_{\text{comm}}}{\mu_{\text{comp}}} \]  

(3.6)

Now, because in the test graphs we assign 1-2 successors to each node, the average number of precedence links drawn from every node is \( \frac{3}{2} \). For the test graphs, it is therefore to be expected that

\[ \rho = \frac{3}{2} \cdot \frac{\mu_{\text{comm}}}{\mu_{\text{comp}}} = \frac{3}{2} \cdot \frac{1}{4} \approx 0.38 \]  

(3.7)

if the number of leaf nodes \( \ll N \).

From table 3.1 we observe that for graphs with \( N = 100 \), the expected \( \rho \)-value is, roughly speaking, maintained within the defined parallelism interval. It means that for a high number of nodes, the interconnection topology in the test graphs is almost insensitive for variations in the graph parallelism. For graphs with \( N = 50 \) and \( N = 25 \), we see decreasing \( \rho \)-values for increasing graph parallelism. In particular for \( N = 25 \), \( \rho \) is only one half of the expected value when the graph parallelism equals 10. This observation basically indicates that for graphs with a small number of nodes (as compared to the degree of parallelism), it is reasonable to expect that \( \rho \ll 0.38 \). One may argue that such graphs are \textit{imbalanced} in the sense that nodes near the bottom of the graph on the average have more input arcs as compared to nodes near the top of the graph.

### 3.5.2 Discussions on the Communication Overheads

So far, we have only specified the \textit{number of data tokens} to be associated to each precedence link. For different types of architectures, the "one-token no-contention communication time" \( t_{\text{token}} \) may vary substantially leading to different performances. In the following experiments we therefore select \( t_{\text{token}} \) equal to 2, 4, and 8, respectively. This means that the total average communication time when executing a graph is 1) half, 2) same as, and 3) double the total average computation time. Defining the communication/computation ratio in a graph \( G \) as

\[ CCR \equiv \frac{t_{\text{token}} \cdot \mu_{\text{comm}}}{\mu_{\text{comp}}} \]  

(3.8)

we get \( CCR = 0.5 \), \( CCR = 1 \), and \( CCR = 2 \) for \( t_{\text{token}} = 2 \), 4, and 8, respectively.

Since we have restricted NODUST to apply to message passing architectures only, we have to decide on the DMA- and setup times (see figure 2.12). We assume, however, that these parts of the overall communication time are equal to zero. With these assumptions we schedule onto fully connected 2, 4, and 6 processor target architectures.

The scheduling results for \( Hu^* \), NODUST/ASAP, and NODUST/CP can be found in tables 3.2, 3.3, and 3.4, respectively.

In order to get an initial overview of these results, we first of all make a simple comparison between \( Hu^* \) and the two NODUST versions for the different \( t_{\text{token}} \) values. The result of this comparison can be seen in table 3.5 where "\( \cdot \)" and "\( \cdot \)" indicate better and worse speedup performances, respectively, as compared to \( Hu^* \).

Now, for each value of the communication/computation ratio \( (CCR=0.5, 1 \text{ and } 2) \), we count the number of times the following cases occur (table 3.6).

Case 1: \( \text{Speedup}_{\text{NODUST/ASAP}} \geq \text{Speedup}_{Hu^*} \)

Case 2: \( \text{Speedup}_{\text{NODUST/CP}} \geq \text{Speedup}_{Hu^*} \)
### Table 3.2: The speedups derived using the Hu⁺ strategy on our defined sets of randomly generated test graphs.

<table>
<thead>
<tr>
<th>Set #</th>
<th>( CCR = 0.5 )</th>
<th>( CCR = 1 )</th>
<th>( CCR = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( p=2 )</td>
<td>( p=4 )</td>
<td>( p=6 )</td>
</tr>
<tr>
<td>1</td>
<td>1.711</td>
<td>1.958</td>
<td>1.958</td>
</tr>
<tr>
<td>2</td>
<td>1.949</td>
<td>3.495</td>
<td>4.778</td>
</tr>
<tr>
<td>4</td>
<td>1.867</td>
<td>2.756</td>
<td>2.791</td>
</tr>
<tr>
<td>5</td>
<td>1.967</td>
<td>3.833</td>
<td>5.423</td>
</tr>
<tr>
<td>7</td>
<td>1.879</td>
<td>3.166</td>
<td>3.510</td>
</tr>
<tr>
<td>8</td>
<td>1.963</td>
<td>3.723</td>
<td>5.248</td>
</tr>
<tr>
<td>9</td>
<td>1.961</td>
<td>3.833</td>
<td>5.600</td>
</tr>
</tbody>
</table>

### Table 3.3: The speedups derived using the NODUST/ASAP strategy on our defined sets of randomly generated test graphs.

<table>
<thead>
<tr>
<th>Set #</th>
<th>( CCR = 0.5 )</th>
<th>( CCR = 1 )</th>
<th>( CCR = 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( p=2 )</td>
<td>( p=4 )</td>
<td>( p=6 )</td>
</tr>
<tr>
<td>1</td>
<td>1.497</td>
<td>1.893</td>
<td>1.938</td>
</tr>
<tr>
<td>3</td>
<td>1.992</td>
<td>3.877</td>
<td>5.521</td>
</tr>
<tr>
<td>4</td>
<td>1.774</td>
<td>2.361</td>
<td>2.661</td>
</tr>
<tr>
<td>5</td>
<td>1.946</td>
<td>3.560</td>
<td>4.807</td>
</tr>
<tr>
<td>7</td>
<td>1.808</td>
<td>2.731</td>
<td>3.102</td>
</tr>
<tr>
<td>8</td>
<td>1.955</td>
<td>3.531</td>
<td>4.680</td>
</tr>
<tr>
<td>9</td>
<td>1.969</td>
<td>3.758</td>
<td>5.262</td>
</tr>
</tbody>
</table>
Table 3.4: The speedups derived using the NODUST/CP strategy on our defined sets of randomly generated test graphs.

Table 3.5: Hu* is compared against NODUST/CP (“C”) and NODUST/ASAP (“A”), respectively. A “+” denotes that a speedup improvement is observed as compared to Hu*. In all other cases, the speedup is either equal to (“0”) or lesser than (“-”) the speedup provided by Hu*.
### Quantitative evaluation

<table>
<thead>
<tr>
<th></th>
<th>CCR = 0.5</th>
<th></th>
<th>CCR = 1</th>
<th></th>
<th>CCR = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>4</td>
<td>Case 2</td>
<td>23</td>
<td>Case 3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case 1</td>
<td>5</td>
<td>Case 2</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case 1</td>
<td>16</td>
<td>Case 2</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 3.6: For each value of the one-token communication time, 27 experiments have been performed for Hu* as well as for both versions of the NODUST algorithm. This table indicates the number of experiments which fall within the defined cases 1, 2, and 3.

### ∆% performance between Hu* and NODUST/CP

<table>
<thead>
<tr>
<th></th>
<th>CCR = 0.5</th>
<th></th>
<th>CCR = 1</th>
<th></th>
<th>CCR = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set #</td>
<td>p=2</td>
<td>p=4</td>
<td>p=6</td>
<td>p=2</td>
<td>p=4</td>
</tr>
<tr>
<td>1</td>
<td>2.34</td>
<td>0.66</td>
<td>0.66</td>
<td>-1.37</td>
<td>2.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-0.43</td>
<td>0.33</td>
<td>1.76</td>
<td>2.02</td>
<td>3.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.64</td>
<td>0.98</td>
<td>0.00</td>
<td>1.61</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.56</td>
<td>2.49</td>
<td>3.29</td>
<td>2.90</td>
<td>-1.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.20</td>
<td>-0.29</td>
<td>1.35</td>
<td>-1.22</td>
<td>-0.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.10</td>
<td>2.07</td>
<td>2.53</td>
<td>3.09</td>
<td>1.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.00</td>
<td>-0.70</td>
<td>0.71</td>
<td>2.81</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.25</td>
<td>-0.46</td>
<td>0.18</td>
<td>7.26</td>
<td>-2.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.20</td>
<td>0.31</td>
<td>0.63</td>
<td>-0.79</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: The percentage performance difference between Hu* and NODUST/CP. Positive numbers indicate the percentage improvement of NODUST over Hu.

Case 3: \( \text{Speedup}_{\text{NODUST/ASAP}} \geq \text{Speedup}_{\text{NODUST/CP}} \)

(assuming that \( \text{Speedup}_{\text{NODUST/ASAP}} \geq \text{Speedup}_{\text{Hu*}} \))

From the figures in table 3.6 it is clear that NODUST/CP in most cases is superior to both Hu* and NODUST/ASAP. One exception, though, is for a high communication rate (CCR = 2). In this situation, NODUST/ASAP is better in more than one third of the cases. However, in order to draw some conclusions, we limit ourselves in the following to the NODUST/CP results. Basically, we show the percentage comparison between NODUST/CP and Hu*, table 3.7.

### 3.5.3 The Impact of Changing CCR-values

Analyzing the percentage speedup difference between NODUST/CP and Hu*, one observation is immediate; for CCR = 0.5, the performance improvement for NODUST/CP as compared to Hu* is somewhat limited (typically less than one percent). This result, in fact, is not surprising because NODUST is basically designed to work on graphs with substantial communication and may therefore not perform any better than Hu* in low communication environments.

For CCR = 1, NODUST/CP is in most cases able to generate better solutions as compared to Hu* (typically a few percent). However, from the results it also seems as NODUST performs best when the target architecture has a high number (6) of processors. This may be explained by the
### Table 3.8: Performance of Nodes

<table>
<thead>
<tr>
<th># of nodes</th>
<th>CCR = 0.5</th>
<th>CCR = 1</th>
<th>CCR = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 2</td>
<td>p = 4</td>
<td>p = 6</td>
<td>p = 2</td>
</tr>
<tr>
<td>25</td>
<td>13</td>
<td>95</td>
<td>188</td>
</tr>
<tr>
<td>50</td>
<td>7</td>
<td>41</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>20</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 3.8: This table shows the percentage speedup improvement observed for graphs with a high degree of parallelism as compared to graphs with a low inherent parallelism (having the same number of nodes). The results are obtained using the NODUST/CP strategy (a "-" indicates a performance decrease).

The fact that for equal average communication and average computation, the MIN/MAX problem may only become severe when a substantial part of the inherent graph parallelism is employed by adding more processors to the architecture. Since NODUST tries to solve the MIN/MAX problem, this is a reasonable explanation for the better performance for $p = 6$ as compared to $p = 2$ and $p = 4$.

Finally, for $CCR = 2$, we can safely say that NODUST is a more optimal heuristic as compared to $Hu^*$, although there seems to be a performance difference due to the number of processors. For heavy communication, NODUST performs relatively better on target architectures with a small number of processors. One explanation may be due to the $Hu^*$ selection facility which basically chooses the processor on which one or more immediate predecessors (to $n_j$) are already scheduled (see algorithm 6). It means that $Hu^*$ tries to utilize as few processors as possible and thus limiting the necessary IPC — i.e., $Hu^*$ produces relatively good schedules when a high number of processors are available in the architecture. On a few-processor architecture, however, NODUST may still outperform the $Hu^*$ schedule because of further IPC-reduction obtained by the means of node duplication.

#### 3.5.4 The Impact of Changing Graph Parallelism

We now turn into a brief analysis of the impact on the speedup performance caused by the graph parallelism. Basically, we calculate the percentage speedup improvement when going from the lowest degree of parallelism to the highest for a fixed number of graph nodes (i.e., from graph set 1 to 3, from 4 to 6, and from 7 to 9). These calculations, which are performed for the three different CCR values and the various number of processors, can be seen in table 3.8 (for the NODUST/CP algorithm). The results corresponding to $Hu^*$ show exactly the same overall picture (and are therefore excluded).

Roughly speaking, for $CCR = 0.5$ and for any number of processors, the speedup improvement is decreased by a factor of two when the number of graph nodes is doubled. For $CCR = 1$ and $CCR = 2$ we observe the same phenomenon, although the tendency is less obvious.

The overall explanation may be the following. For graphs with a relatively low number of nodes, the interconnection topology varies substantially according to the inherent parallelism, cf., the $p$-values. Therefore, it is reasonable to believe that for small graphs we will observe a higher change in speedup as compared to larger graphs when the parallelism is increased. This is consistent to our performance results, although it is somewhat difficult to give a solid explanation of the speedup improvements in relation to the actual interconnection topology ($p$) values.
3.6 Dynamic Level Scheduling

Sih & Lee, [42], proposed the very nice and simple Dynamic Level Scheduling (DLS) approach. Due to the conclusions drawn in [43] which state that DLS is among the best list scheduling heuristics known today (Liao et al. investigated various strategies derived from the DLS approach, e.g., differences in node and processor selection), we in this section very briefly outline the overall principle of the DLS concept and next compare it to our NODUST strategy (we note that NODUST apparently was unknown to the authors of [43] and therefore it was not part of their evaluation).

Sih & Lee noted a severe problem which characterizes HLF based techniques. In principle, HLF selects the node (from the priority level), which is currently in the critical path. Next, it assigns this node (arbitrarily) to an available processor (a processor which, according to a global scheduling clock, is currently idle). The problem here is that such a processor may not necessarily — due to the IPC time — be the processor which (from a global point of view) matches the node most optimally. An obvious solution to this problem is (like we also suggested for Hu* and NODUST) to make an evaluation of all processors instead of only searching through available processors.

Now, the earliest possible invocation time of a given node \( n_j \) on a particular processor \( p_k \) is determined by the following parameters:

- the time when \( p_k \) finishes its current task — \( TF \) (time finish)
- the time when all necessary data tokens for \( n_j \) are available on \( p_k \) — \( DA \) (data available)

Both \( TF \) and \( DA \) are dependent on the current state of the communication facilities, i.e., the ability to communicate data tokens. Sih & Lee denoted the state of the communication hardware \( \Sigma \).

They next defined a Dynamic Level (DL) as

\[
DL(n_j, p_k, \Sigma) \equiv SL(n_j) - \max\{DA(n_j, p_k, \Sigma), TF(p_k, \Sigma)\}
\] (3.9)

In equation 3.9, \( SL \) denotes the static level of node \( n_j \) — i.e., the location of \( n_j \) in the critical path (see algorithm 2). The second term includes \( DA \) and \( TF \) and it selects the maximal of these two values. This is done due to the fact that \( n_j \) cannot be invoked on \( p_k \) before it has terminated its previous task (see the PWC model, section 2.3), and similarly, \( n_j \) cannot be invoked on \( p_k \) before all necessary data tokens are available.

The idea now is that for “a number” of executable nodes and processors \( (n_j, p_k) \) the DL-values are calculated, and the pair which shows the highest dynamic level, i.e., the pair which has the best match, is chosen. The reason for selecting the highest DL-value is that we want to schedule a node with as high a location in the critical path as possible, i.e., a node with a high priority for execution (MAX \( SL \)). At the same time we want to choose a processor which can invoke the node as early as possible, i.e., a processor where data are early available (MIN \( \max\{DA, TF\} \)).

Concerning the node/processor pair, Sih & Lee proposed three different approaches.

- Select an executable node and evaluate all processors
- Select a processor and evaluate all executable nodes
- Evaluate all pairs of executable nodes and processors
CHAPTER 3. LIST SCHEDULING HEURISTICS

<table>
<thead>
<tr>
<th>Set #</th>
<th>CCR = 0.5</th>
<th>CCR = 1</th>
<th>CCR = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=2</td>
<td>p=4</td>
<td>p=6</td>
</tr>
<tr>
<td>1</td>
<td>1.788</td>
<td>1.958</td>
<td>1.958</td>
</tr>
<tr>
<td>4</td>
<td>1.899</td>
<td>2.799</td>
<td>2.840</td>
</tr>
<tr>
<td>5</td>
<td>1.985</td>
<td>3.867</td>
<td>5.571</td>
</tr>
<tr>
<td>6</td>
<td>1.988</td>
<td>3.918</td>
<td>5.731</td>
</tr>
<tr>
<td>7</td>
<td>1.916</td>
<td>3.255</td>
<td>3.528</td>
</tr>
<tr>
<td>8</td>
<td>1.977</td>
<td>3.850</td>
<td>5.541</td>
</tr>
<tr>
<td>9</td>
<td>1.989</td>
<td>3.889</td>
<td>5.695</td>
</tr>
</tbody>
</table>

Table 3.9: The speedups derived using the Dynamic Level Scheduling strategy on our defined sets of randomly generated test graphs.

It is evident that the search space is larger (and thus the probability of finding an optimal solution is higher) for the latter method as compared to the two initial ones but, of course, at the expense of an increased computational complexity. For an exhaustive explanation of the DLS approach, see [42].

As previously mentioned, researchers have reported on the DLS technique to be among the best known list scheduling technique, and we therefore want to compare our NODUST heuristic to the DLS approach. These investigations are reported below.

Comparing DLS to NODUST

In order to compare the heuristics, we conducted DLS experiments on our sets of randomly generated graphs (table 3.1). Our DLS implementation performs the exhaustive search where dynamic levels for all pairs of executable nodes and processors are calculated. Due to the DA value required by the DLS algorithm, all communication from predecessors have to be prescheduled — in contrast to Hu* and NODUST where we do not consider communication prescheduling. Basically, we extract a copy of the actual communication schedule on which we next do all the prescheduling as needed in order to determine the DA values.

The average speedups for the nine sets are shown in table 3.9. From these figures we next extract the percentage speedup improvements as corresponding to increased parallelism for maintained number of graph nodes, table 3.10. The table reflects basically the same information as observed for the NODUST algorithm, and thus the overall conclusion is the same. Finally, table 3.11 gives the percentage performance difference between NODUST/CP and DLS.

Referring to table 3.11, the overall message (from our point of view) is somewhat disappointing. DLS is outperforming NODUST in most cases. For a small communication overhead, the improvement gained by using DLS, however, is quite limited — but still it is typically 1-2 %. For CCR=1, on the other hand, it seems as DLS is able to generate schedules which are significantly better than NODUST schedules. We hardly can point out a specific situation where NODUST shows better performance. The same is more or less true for CCR=2, the only exception being for graphs with a small number of nodes — here NODUST is best in 6 out of 9 possible cases.

However, before we draw any definite conclusions towards an absolutely honoring of DLS as
### CHAPTER 3. LIST SCHEDULING HEURISTICS

#### Speedup Improvement (%), Low → High Parallelism

<table>
<thead>
<tr>
<th># of nodes</th>
<th>$CCR = 0.5$</th>
<th>$CCR = 1$</th>
<th>$CCR = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=2</td>
<td>p=4</td>
<td>p=6</td>
</tr>
<tr>
<td>25</td>
<td>11</td>
<td>63</td>
<td>190</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>40</td>
<td>102</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>19</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 3.10: This table shows the percentage speedup improvement observed for graphs with a high degree of parallelism as compared to graphs with a low inherent parallelism (having the same number of nodes). The results are obtained using the DLS strategy (the "-" indicates a performance decrease).

#### Δ% performance between NODUST/CP and DLS

<table>
<thead>
<tr>
<th>Set #</th>
<th>$CCR = 0.5$</th>
<th>$CCR = 1$</th>
<th>$CCR = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=2</td>
<td>p=4</td>
<td>p=6</td>
</tr>
<tr>
<td>1</td>
<td>2.11</td>
<td>-0.66</td>
<td>-0.66</td>
</tr>
<tr>
<td>2</td>
<td>0.31</td>
<td>-0.67</td>
<td>0.61</td>
</tr>
<tr>
<td>3</td>
<td>0.30</td>
<td>-17.01</td>
<td>0.04</td>
</tr>
<tr>
<td>4</td>
<td>2.15</td>
<td>1.23</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0.71</td>
<td>1.12</td>
<td>1.36</td>
</tr>
<tr>
<td>6</td>
<td>-0.20</td>
<td>0.44</td>
<td>1.02</td>
</tr>
<tr>
<td>7</td>
<td>1.32</td>
<td>1.81</td>
<td>0.51</td>
</tr>
<tr>
<td>8</td>
<td>0.61</td>
<td>1.32</td>
<td>2.97</td>
</tr>
<tr>
<td>9</td>
<td>1.22</td>
<td>1.14</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Table 3.11: The percentage performance difference between NODUST/CP and DLS. Positive numbers indicate the percentage improvement of DLS over NODUST.
CHAPTER 3. LIST SCHEDULING HEURISTICS

<table>
<thead>
<tr>
<th>Set #</th>
<th>NODUST/CP p=2</th>
<th>p=4</th>
<th>p=6</th>
<th>DLS p=2</th>
<th>p=4</th>
<th>p=6</th>
<th>Δ % p=2</th>
<th>p=4</th>
<th>p=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.680</td>
<td>0.856</td>
<td>0.858</td>
<td>0.396</td>
<td>0.572</td>
<td>0.589</td>
<td>71.7</td>
<td>49.7</td>
<td>45.7</td>
</tr>
<tr>
<td>2</td>
<td>0.678</td>
<td>1.118</td>
<td>1.219</td>
<td>0.498</td>
<td>0.891</td>
<td>1.112</td>
<td>36.1</td>
<td>25.5</td>
<td>9.6</td>
</tr>
<tr>
<td>3</td>
<td>1.454</td>
<td>1.885</td>
<td>1.290</td>
<td>0.845</td>
<td>1.022</td>
<td>1.271</td>
<td>72.1</td>
<td>84.4</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>0.518</td>
<td>0.886</td>
<td>0.890</td>
<td>0.374</td>
<td>0.685</td>
<td>0.810</td>
<td>38.5</td>
<td>29.3</td>
<td>9.9</td>
</tr>
<tr>
<td>5</td>
<td>0.603</td>
<td>1.138</td>
<td>1.467</td>
<td>0.332</td>
<td>0.869</td>
<td>1.326</td>
<td>81.6</td>
<td>31.0</td>
<td>10.6</td>
</tr>
<tr>
<td>6</td>
<td>0.776</td>
<td>1.054</td>
<td>1.453</td>
<td>0.432</td>
<td>0.978</td>
<td>1.451</td>
<td>79.6</td>
<td>7.8</td>
<td>0.1</td>
</tr>
<tr>
<td>7</td>
<td>0.454</td>
<td>0.903</td>
<td>1.138</td>
<td>0.463</td>
<td>0.869</td>
<td>1.020</td>
<td>-1.9</td>
<td>3.9</td>
<td>11.6</td>
</tr>
<tr>
<td>8</td>
<td>0.359</td>
<td>0.800</td>
<td>1.394</td>
<td>0.324</td>
<td>0.807</td>
<td>1.353</td>
<td>10.8</td>
<td>-0.9</td>
<td>3.03</td>
</tr>
<tr>
<td>9</td>
<td>0.360</td>
<td>0.800</td>
<td>1.395</td>
<td>0.361</td>
<td>0.890</td>
<td>1.376</td>
<td>-0.3</td>
<td>-10.1</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 3.12: The speedup measures for NODUST/CP and DLS in the presence of a very high communication overhead. Furthermore, the table gives the percentage performance improvement of NODUST/CP over DLS (i.e., negative numbers represent cases where DLS is outperforming NODUST/CP). The emphasized numbers indicate the interesting situations where NODUST/CP is generating speedups > 1.

The ultimate static list scheduling technique for data independent scientific computations, we propose to conduct further experiments with more severe communication overheads. Beside, an evaluation based on real DSP DAPGs may also shown some different result. This is left for chapter 6.

3.7 Scheduling In the Presence of Heavy Communication

Some types of DSP algorithms may be characterized by a very high data throughput, and at the same time they have a limited computational overhead, examples are image- and radar front-end algorithms.

Since this type of applications are within the scope of our research domain, it is relevant to investigate their expected scheduling performance.

In this section, we therefore present scheduling results obtained using the NODUST/CP and DLS strategies on the random test graph sets. The communication overhead, however, has be substantially increased as compared to our previous experiments. We have now selected $t_{token} = 40$, implying that CCR equals 10. Table 3.12 present the actual speedup measures for our test sets according to this communication overhead.

From the table it is clear that the severe communication load is often causing the two scheduling strategies to produce solutions with a speedup lesser than 1. These cases are of no interest since the sequential solution, of course, is more optimal. We observe for the complete set of experiments, although, that 1) NODUST/CP is generating more solutions with an "above 1" speedup as compared to DLS, and 2) in all those cases where NODUST/CP is producing a speedup greater than 1, the corresponding DLS solution always has a lower speedup (indicated by emphasized positive numbers).

Based on these experiments, we therefore have reasons to believe that multiprocessor scheduling of applications imposed by a severe communication overhead can be done more efficiently by the NODUST/CP strategy as compared to what may be obtained using the DLS approach.
An argument here, however, is that a speedup, of let's say 1.3 on a 6-processor architecture (which seems to be a typical result), is a too low performance gain as related to the cost and complexity of such a system. On the other hand, there exists applications with high CCR-values for which it in some cases may be necessary to propose parallel solutions having a 30% performance improvement over the sequential solution. Therefore, we have confidence in the profitable usage of NODUST/CP in such cases.

Taking a closer look at the scheduling results we also observe that in most cases, the percentage improvement of NODUST/CP over DLS is decreasing as the number of processors in the target architecture is increased. The reason for this phenomenon is likely to be the load balancing nature of NODUST/CP which is not characterizing DLS — remember that DLS is searching for the best node/processor match. The reason for the better performance obtainable using NODUST/CP is therefore due to the duplication of one or more nodes in order to minimize the total communication overhead.

In this evaluation we have not considered the processor utilization factor, but there is no doubt that for the very high communication overheads employed in these experiments, the degree of processor utilization is relatively low. Since we are mainly interested in the speedups, the utilization will not be paid further attention.
Chapter 4

Static Scheduling using Simulated Annealing

In chapter 3 we presented suggestions for realistic and efficient list scheduling strategies to be included in our Multiprocessor Scheduling Pool. Through a set of experiments performed on randomly generated test graphs, we concluded that the NODUST approach is the most optimal one of the techniques proposed by us.

We also showed that other researchers have done an excellent job in order to develop a simple and highly efficient strategy — the Dynamic Level Scheduler — which actually for “low”, “modest”, and “high” communication rates in most cases is able to produce better schedules as compared to the ones generated by NODUST. However, for algorithms bounded by a “ultra-high” communication overheads, we found that NODUST is always doing better than DLS.

These results obviously indicate that DLS cannot be honored as the ultimate scheduling strategy and we have therefore been encouraged to search for other scheduling techniques which in general are able to further optimize the DLS solution.

Based on the conclusions drawn in [43] which states that the DLS strategy is among the best known scheduling heuristics of today, it seems hard to believe that we can devise a new list scheduler with even better performance. On the other hand, since the scheduling problem has NP-complete complexity and the DLS strategy searches only through parts of the solution space, we may be able to develop an alternative method with improved performance if we allow a more thorough search.

In light of the fact that the Hu*-NODUST-, or DLS schedule of a 100 node DAPG onto a 6 processor architecture is obtained in a few seconds using a SPARC10 workstation, we safely allow a substantially higher scheduler complexity.

It is therefore obvious that we should turn our attention into iterative methods which, as compared to the general list scheduling approach, are able to search larger parts of the solution space. One possible method which is known to suit combinatorial optimization problems is Simulated Annealing (SA), [44].

This chapter is therefore devoted to our attempts towards developing SA-based static scheduling techniques. In order to prepare the reader for a clear understanding of our SA strategies, we initially review the basics of simulated annealing.
4.1 Fundamentals of Simulated Annealing

In general, a combinatorial optimization problem may have an enormous solution space \( S_{\text{total}} \) where the \emph{optimal} solution (to be called \( s_{\text{opt}} \) hereafter) normally can only be found by employing an exhaustive search. For even a limited problem complexity, this prohibits the optimal solution from being detected in a reasonable time. As we have also seen in chapter 3, heuristics which try to locate \( s_{\text{opt}} \) in “one step” suffer from the \emph{horizon effect} and therefore the proposed solution is only \emph{near optimal}.

Intuitively, it makes sense to conceive \( S_{\text{total}} \) as a set of solutions located next to each other with neighbour solutions being those which are “close” in some sense. Taking this approach, which we illustrate in figure 4.1, one can (so to speak) “walk around” in the solution space — from one solution to the next. In order to arrive at \( s_{\text{opt}} \) from the solution at which we are currently staying, we therefore have to follow a path (it is reasonable to assume that there are more than one) leading towards \( s_{\text{opt}} \), i.e., we have to generate a number of solutions which will lead exactly or very close to the optimal solution.

![Solution space](image)

Figure 4.1: The solution space can be seen as a finite set of solutions. Applying a well defined set of changes to the current solution \( i \), it is possible to visit a subset of the solution space defined as the neighbourhood of \( i \).

Staying at a particular current solution \( s_i \), we define the “neighbourhood of \( s_i \)” as the set of solutions \( \{s_j\} \) which can be reached by performing a change to \( s_i \). By “change” we mean a well defined set of operations applied to the current solution. Taking the famous \emph{Traveling Salesman Problem} as an example, \cite{45}, a change would be to swap two cities.

In our case of static multiprocessor scheduling, we can consider a schedule (assignment, ordering, and invocation time specification) as a \emph{solution}. Obviously, the optimal solution (i.e., the optimal schedule in terms of minimal makespan), corresponds to a specific allocation of nodes onto the processors (we assume that the optimal schedule corresponds to one and only one assignment/ordering for the given target architecture). Therefore, the solution space can be explored by generating a series of new assignments/orderings.

Accepting the idea of a solution space in which we through a number of changes can approach towards \( s_{\text{opt}} \), the problem now is to define a strategy which actually will lead us in the right direction. Since to every solution there can be associated a \emph{cost} (e.g., makespan), the technique proposed by Kirkpatrick et al., \cite{46}, known as \emph{Simulated Annealing} seems fairly obvious.

The basic principle of this method is to emulate the way a heated and melted material through a cooling process is brought into a solid. When the material is heated sufficiently, it enters a liquid phase where the particles are placed randomly. This corresponds to a \emph{high energy state}. 


Lowering the temperature (slowly), the material will turn into a low energy state corresponding to a solid with its particles arranged in a highly structured lattice — this is known as thermal equilibrium. From a physical point of view, this annealing process is indeed very complicated and a detailed discussion is therefore out of scope in our context. However, it should be noted that the annealing should take place in a slowly and controllable manner. Although it will bring the material into a solid, an instantaneous lowering of the temperature will also cause the material to enter an unstable state because it will never reach its low energy state. This type of cooling is known as quenching.

4.1.1 The Metropolis Criterion

In order to simulate the annealing process, it is assumed that a sequence of states is generated which represent specific orderings of the particles in the material. The transition from one state to the next is obtained by applying a perturbation mechanism — e.g., changing the position of a particle. To each state is associated a cost in terms of energy.

Let the current state be denoted \(i\) and the next state \(j\). Therefore, by the perturbation mechanism, the energy is changed from \(E_i\) to \(E_j\). If \(E_j \leq E_i\), we are approaching towards thermal equilibrium and the next state is therefore accepted — i.e., the next state becomes the current state. If, on the other hand, \(E_j > E_i\), we have actually generated a next state characterized by a higher cost. According to our discussion above, assume there is a path of states (solutions) through which we have to travel in order to reach \(s_{opt}\). However, we have not said anything about the costs of the individual solutions along that path. It may very well be that we have to perform some up-hill moves (in terms of accepting next solutions with higher cost). From that perspective, we should allow hill-climbing to occur, but the probability for a cost increase to be accepted should be controlled in some manner — otherwise, we would “walk around” with no guidance at all. The Metropolis criterion gives the answer to the acceptance problem for cost increasing solutions.

\[
P(\text{accept } j) = \exp\left(\frac{E_i - E_j}{k_B T}\right)
\]  

(4.1)

Here \(T\) is the actual temperature and \(k_B\) is the Boltzmann constant. Obviously, the higher the cost increase and the lower the temperature, the less likely the acceptance of the generated cost increasing solution.

The fundamental SA algorithm is directly derived from equation 4.1. However, since a combinatorial optimization problem to be solved by SA normally have no immediate relations to the cooling of a heated material it doesn’t make sense to talk about “the temperature”. Therefore, the denominator is substituted by a control parameter, \(c\), which initially has a “sufficiently” high value and then is “slowly” decreased.

Simulated Annealing is then defined, for each value of the control parameter, as a number of perturbations (to be called transitions henceforth) evaluated by the Metropolis criterion. This is illustrated by algorithm 8, where \(c_{\text{init}}\) and \(c_{\text{stop}}\), are the start and stop values for the control parameter, respectively. In this equation, \(L\), corresponds to the number for transitions for each control parameter value. Note that the acceptance of a cost increasing solution is formulated in terms of a comparison against a randomly generated number in the interval from 0 to 1. This reflects the actual way of implementing the acceptance probability function where we assume a uniform distribution of the random generator.
\begin{algorithm}
\begin{enumerate}
\item $s_i$ represents the initial solution;
\item $k = 0$;
\item $c_k = c_{init}$;
\item While $c_k > c_{stop}$ do
\begin{enumerate}
\item For $l = 1$ to $L$
\begin{enumerate}
\item generate $s_j$ from $s_i$;
\item if $\text{cost}(s_j) \leq \text{cost}(s_i)$ then $s_i = s_j$;
\item else if $\exp((\text{cost}(s_i) - \text{cost}(s_j))/c_k) > \text{random}[0;1]$ then $s_i = s_j$;
\end{enumerate}
\item $k := k + 1$;
\item calculate $c_k$;
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{algorithm}

\textbf{Algorithm 8:} The fundamental nature of Simulated Annealing expressed by pseudo code — $L$ evaluated transitions for every control parameter value $c_k$.

4.1.2 Essential Elements in SA

From algorithm 8 it is clear that a SA algorithm must be specified in terms of various characteristic features. Basically, the following is required.

- An initial value of the control parameter, $c_{init}$
- A final value for the control parameter, $c_{final}$
- A cooling strategy $C$; $c_k = C(c_{k-1})$
- A fixed number of transitions $L$ (possibly variable as a function of the control parameter step, $L_k$)
- A transition function $T$; $s_j = T(s_i)$

The value $c_{init}$

Concerning the initial control parameter value, its value must be sufficiently high to represent the cost of the initial solution — which is expected to be rather high too, due to its random nature. Therefore, we need an algorithm for calculating an appropriate $c_{init}$.

One commonly employed strategy is to use an \textit{acceptance ratio}, $\chi(c)$, defined as

$$\chi(c) = \frac{\text{number of accepted transitions}}{\text{number of generated transitions}}$$

Equation 4.2 expresses for a given value of $c$ the number of accepted transitions related to the number of generated transitions. Turning this equation upside down, it can be used to find the $c$-value for which, say 95-99\% of all proposed solutions are accepted.

So, the strategy simply is to increase $c$ until a prespecified acceptance ratio is reached. This corresponds to a \textit{heating process} — at a sufficiently high control parameter value, nearly all proposed solutions will be accepted. This is a desired situation because we, for \textit{that} $c$-value, are able to accept almost any solution in $S_{total}$ and thus we have not precluded any solution from being visited and evaluated.
CHAPTER 4. STATIC SCHEDULING USING SIMULATED ANNEALING

The value \( c_{\text{stop}} \)

The final control parameter value corresponds to the value at which the SA process terminates. This means basically that \( c_{\text{stop}} \) can be interpreted in two different ways.

- The search terminates when \( c \) is decreased to a prespecified value, i.e., \( c_{\text{stop}} \) is the stop criterion.
- The search terminates when a "c-independent" stop criterion is fulfilled, i.e., \( c_{\text{stop}} \) is simply the control parameter value upon termination.

Using a predefined \( c_{\text{stop}} \)-value as the stop criterion (as shown in algorithm 8) has the advantage that (given \( i \)) a known number of transitions for each control parameter value and \( ii) \) a cooling strategy which depends only on \( c \) termination will occur after a known number of transitions. The drawback is that we for a fixed number of transitions may still be far from the optimal solution.

Alternatively, the stop criterion could relate to the performance of the solutions (in terms of cost). Assuming for instance that we want to detect a solution characterized by a given cost, we simply keep the SA algorithm running until such a solution is found. Here the drawback is that we do not know when the SA search will converge.

The cooling strategy

The strategy applied in order to decrease the control parameter is essential for the performance for the overall search. As we have already mentioned, the possibility for quenching is impending if \( c \) is decreased too rapidly. On the other hand, lowering \( c \) extremely slow will prohibit quenching, but it will most likely also lead to an unacceptable long run time.

Therefore, the cooling strategy must be designed to trade off the schedule quality against necessary run time.

Designing a SA algorithm, there are several options as related to the choice of cooling strategy. In our opinion, these possibilities can be divided into two major categories:

- strategies which are functions of only \( c \)
- strategies which are functions of \( c \) and other variables

Strategies of multiple variables can be made adaptive in the sense that step size is adjustable due to e.g., changes in statistics of cost values. This facility is not possible in strategies which employ \( c \) as the only variable.

The number of transitions

For every value of the control parameter we must generate, evaluate and accept/reject a number of transitions before we go to the next value of \( c \).

Since each transition generates the next solution based on the current solution, the sequence of transitions for any \( c \)-value can be modeled mathematically by a Markov chain. It means that for the current solution, \( s_i \), the generation of the next solution can be seen as a selection (with a certain probability) of a solution \( s_j \) belonging to the neighbourhood of \( s_i \). Since the neighbourhood of \( s_i \) is a finite set of solutions (including \( s_i \) itself), the model more precisely is a finite Markov chain.
CHAPTER 4. STATIC SCHEDULING USING SIMULATED ANNEALING

It is evident that the number of transitions for a given $c$-value equals the length of the corresponding Markov chain. The question now is; how long a Markov chain (i.e., the value of $L$) is required? It is out of the scope in this thesis to conduct the proof, but referring to [44] it can be shown that a SA algorithm has asymptotic convergence. It basically means that SA is guaranteed to find an optimal solution if the length of the Markov chains are infinite long. This can be expressed mathematically as

$$\lim_{c \to 0} \lim_{l \to \infty} P_c \{X(l) \in S_{opt}\} = 1 \tag{4.3}$$

where $X(l)$ is a stochastic variable denoting the outcome (i.e., solution) of the $l^{th}$ transition, and $S_{opt}$ is the set of optimal solutions (in general we assume that a combinatorial optimization problem may have multiple optimal solutions).

Clearly, this is of no practical use and we therefore have to relax the Markov chain length requirement. The consequence, of course, is that finite length Markov chains cannot guarantee optimal convergence.

Therefore, it is a matter of experimental experiences to decide upon the Markov chain length for a given type of combinatorial optimization problem.

The transition function

All the SA related items discussed above have to be tuned to the combinatorial optimization problem to be solved. However, no matter the problem, their implementations are roughly speaking invariant. This is particularly not the case of the transition function which have to be designed with the specific application in mind.

First of all, the combinatorial optimization problem has to be formulated so that it can efficiently be represented by dedicated data structures, and next the transition function must be designed to operate on these data structures. From our point of view, this is actually the major task in designing a SA algorithm for a given problem.

4.2 Pioneers in SA Based Scheduling

Before we go into a detailed discussion of our SA scheduling techniques, we note initially that other researchers previously have applied simulated annealing to solve related scheduling problems. In particular, [47], [48], and [49] have discussed SA as a tool for scheduling tasks onto multicomputer networks.

In the paper by Bollinger, [47], it is assumed 1) that the number of tasks is less than or equal to the number of machines, and that 2) the tasks are arranged by an undirected graph. The purpose of the mapping technique is to minimize 1) the total amount of IPC and 2) the communication load on the individual links. Due to these restrictive assumptions, the technique is obviously of limited applicability for our scheduling problem.

Hwang, [48], tries to alleviate the restrictions by allowing the number of tasks to be larger than the number of processors, but still he is accepting undirected graphs. It means basically that the job of his SA algorithm is to balance the total work load onto the machines. Since we are interested only in techniques suited for scheduling directed task graphs, Hwang's algorithm is also out of scope.

In order to devise a more general technique, D'Hollander et al., [49], suggested a SA algorithm which allows the mapping of directed graphs with a number of tasks being larger than the number of processors in the network. The object function employed is a weighted sum of the
total communication overhead and a term which relates to the load balancing of the task graph on the architecture.

As compared to our scheduling problem, this SA strategy seems attractive at the first glance. We therefore introduce the overall working of the technique in terms of a pseudo code shown in algorithm 9.

1) Compose a packet \((PA)\) of all idle processors and all ready tasks;
2) FOR \(c = C_{start} \text{ DOWNTO } C_{stop}\) DO
   
   \[
   \begin{align*}
   & \text{select from } PA \text{ arbitrarily a processor } p \text{ and a task } t; \\
   & \text{IF } p \text{ is idle THEN assign } t \text{ to } p; \\
   & \text{ELSE remove the executing task from } p \text{ and substitute by another task from } PA; \\
   & \text{Accept/Reject using the Metropolis criterion;}
   \end{align*}
   \]

3) Repeat from 1) until all task are assigned;

**Algorithm 9 : Outline of the SA based scheduling technique proposed by D'Hollander.**

Immediately, we recognize that D'Hollander did not design his algorithm according to the general SA methodology. For each value of the control parameter, only one new solution is proposed. The paper is not discussing this point at all but we must assume that the control parameter is being decreased extremely slow in order to ensure a sufficiently large number of combinations to be evaluated.

Another point of criticism, which we believe is perhaps even more severe, relates to the task selection strategy. Note that tasks are selected and assigned (successively presumably) to the processors as they (the tasks) become ready, i.e., *tasks are basically selected for assignment due to their appearance in the critical path*. This is comparable to the selection procedure applied in the fundamental HLFET list scheduling heuristic, [40], which is far from optimal in the presence of IPC. That was also observed by Sih & Lee, when they designed the DIS heuristic (see section 3.6). Although it is not discussed explicitly in [49], D'Hollander must have been aware of this fact since he allows an already assigned task to be remove from a processor and substituted by another ready-task (i.e., his do not exclude processors which have been assigned one of the ready nodes).

However, our point here is as follows; *since tasks are assigned successively according to their location in the critical path there will exist valid combinations of assignments/ orderings which are not possible to evaluate and thus the algorithm has an inherit limitation in its search space.*

According to simulation results given by D'Hollander, his algorithm is able to generate schedules which are up to 50% better (in terms of speedup) than schedules derived using the HLFET heuristic. There are several circumstances which complicate an immediate comparison to our results. Basically, these relate to differences in

- object function — makespan versus comm/load-balance
- target architecture — fully connected versus network
- scheduling approach — non-preemptive versus preemptive

Furthermore, D'Hollander do not discuss which IPC HLFET algorithm his is using for comparison.

Therefore, one could argue that we should adapt the principal ideas from D'Hollander's algorithm into a modified version in order to get started with SA based scheduling. Due to the weaknesses of the reduced search space, however, we have decided to design “from scratch” in order to alleviate this problem. Our ideas are discussed below.
4.3 SAS — The Simulated Annealing Scheduler

It goes without saying that our SA based scheduling strategies have to be included in the pool of scheduling techniques. Therefore, of course, they have to comply with the definitions and models as discussed in chapter 2.

With the above discussion of D'Hollander's algorithm in mind, we in particular stress the importance of the SA search to enable exploration of the entire solution space. According to the definitions in section 2.3, we therefore require that a SA based scheduling algorithm $g$ fulfills the equation

$$ S_g = S_{total} \quad (4.4) $$

It basically means that any node must have the ability to

- reside on any processor
- be located at any position on the selected processor

The first requirement is obvious, the second perhaps being a bit more complex. Nevertheless, in order to generate a sequence of schedules with these characteristics, it all boils down to the design of an appropriate transition function.

According to the fundamental working of simulated annealing, we assume that the initial schedule (corresponding to a high value of the control parameter) is random. It means that nodes are assigned arbitrarily to the processors, and that the nodes are ordered arbitrarily on the individual processors. As a consequence, we do not know exactly how many nodes are assigned to the various processors in the initial solution. Therefore, if we opt for a transition function which simply swaps two nodes, then the number of nodes on the individual processors will remain constant throughout the SA process.

In the worst case situations we can think of all nodes initially being assigned to the same processor which prohibit any parallelism from being exploited. Similarly, a load balanced initial solution (in terms of approximately equal number of nodes being assigned to the processors) will remain load balanced, no matter if the optimal solution is the sequential one.

Thus, another more flexible transition function has to be derived. Basically, what we do is to select (in the current solution) a move node (to be called $n_j$ henceforth). The selection of $n_j$ is arbitrary among all nodes.

In the current solution, $n_j$ resides on a processor which we will now define as the donor processor.

In order to make a transition, we next select arbitrarily an acceptor processor. Due to the random selection, we allow the acceptor processor to be equivalent to the donor processor.

The idea now is to move $n_j$ from its current location at the donor processor to a location on the acceptor processor. Therefore, it is obvious that the acceptor processor selection corresponds to assignment and the selection of location on the acceptor corresponds to ordering. This is illustrated in figure 4.2.

For this transition function to work, we define the term “processor entry” which specifies a node's location on the processor — the very first node being located at entry “1”. This is a mechanism required to express the ordering of the nodes on the processors. Nodes assigned to a given processor are arranged by the total ordering $\rightarrow$, where

$$ n_i \rightarrow n_k \quad (4.5) $$
means that $n_i$ has a lower processor entry number as compared to $n_k$.

For all pair of nodes $(n_i, n_k)$ assigned to a given processor, $n_i$ must be invoked before $n_k$ if $n_i \rightarrow n_k$. This, however, does not necessarily mean that a precedence relation exists between $n_i$ and $n_k$ — they may very well be independent.

It should be emphasized that the transition function does not operate on the actual schedule. It works on a pseudo-schedule established in terms of nodes assigned to processor entries. In section 4.3.2 we discuss the procedure employed for transforming the pseudo schedule into a real schedule. First, however, we in brief elaborate on the algorithm we have devised for moving $n_j$.

### 4.3.1 Moving The Selected Node

Once $n_j$ has been selected, it is removed from the donor processor. Unless $n_j$ is the final node, i.e.,

$$\text{if } \forall n_k \in p_{donor} \mid n_j \rightarrow n_k$$

then the movement necessarily gives rise to an update of the processor entries,

$$\forall n_k \in p_{donor} \mid n_j \rightarrow n_k, \text{ the entry of } n_k \text{ is decreased by 1}$$

This procedure ensures that no entries before the final one (which contain a node) are left open on the donor.

After selecting the acceptor, we next have to decide upon the entry for $n_j$. Since nodes on a processor are ordered by the total ordering $\rightarrow$, the entry selection has to maintain the precedence relations. It means that the entry must be selected such that

- all parent nodes to $n_j$ already residing on the acceptor remain located before $n_j$
• all child nodes to \( n_j \) already residing on the acceptor remain located after \( n_j \)

According to these requirements, we must detect the parent node (on \( p_{acceptor} \)) which have the highest entry, and similarly the child node (also on \( p_{acceptor} \)) which have the lowest entry. From these numbers we calculate the slack in which \( n_j \) can reside on the acceptor. This is illustrated on figure 4.3 where \( n_p \) and \( n_c \) represent the latest parent node and the earliest child node, respectively.

![Figure 4.3: Selecting an entry for the move node on the acceptor requires the precedence relations to be maintained.](image)

We illustrate the move of a node by the case shown in figure 4.4 — the SDF- and DAP graphs of a second order IIR section. In this example, \( n_1 \) is arbitrarily selected as the move node in the current solution. It follows immediately, that \( p_1 \) is the donor processor. Removing \( n_1 \) from the donor means that the entries for \( n_7 \) and \( n_5 \) are decreased by 1. The acceptor processor is selected as \( p_0 \), and for this processor we detect the latest parent \( (n_4) \) and the earliest child \( (n_2) \). In this case, the slack for \( n_1 \) is 0, meaning that it can reside in only one entry — the one between \( n_4 \) and \( n_2 \).

The assignment/ordering, of course, do not represent the schedule. Basically three things are still missing,

• expansion of nodes according to estimated run times
• maintaining of interprocessor precedence relations
• insertion of IPC

In the next section we elaborate on these items.

### 4.3.2 From Processor Entry to Schedule

As pointed out, several actions still need to be defined before we have obtained the total transition function. These are discussed below.
Figure 4.4: An example of 1) move node selection, 2) update of donor processor, 3) calculation of slack, and 4) finally assignment/ordering on the acceptor processor.

Node expansion

In the general case we must expect nodes to have various estimated execution times. As a consequence, these times have to be employed in the schedule and we therefore expand the nodes. This is the first step towards the schedule because we are now inserting time into the assignment/ordering — processor entry numbers are of no meaning any longer.

Maintaining the global precedence relations

The second step is basically equivalent to establishing a schedule without IPC. We now take into account the precedence relations among all nodes (on all processors), such that \( n_i \rightarrow n_k \) if \( (n_i, n_k) \in S_o \). In most cases means that we are now introducing some idle time into the processors. The schedule, however, is not realistic until we have also inserted IPC.

Inserting IPC

The final step is devoted to the insertion of interprocessor communication — either in terms of send/receive messages for shared memory topologies or reservation of communication channels for message passing architectures (see sections 2.4 and 2.5). Inserting IPC, of course, is required in order to make the schedule realistic.

Refering to the example in figure 4.4 we present the final schedule in figure 4.5 where nodes are expanded, precedence relations are maintained, and IPC for a message passing architecture
with a “one time unit per token” transfer rate is inserted. In this specific example, we note that the makespan is reduced due to the application of the transition function.

Figure 4.5: Assuming a message passing architecture with a half duplex communication link, the schedules for 1) the current assignment/ordering and 2) after application of the transition function are obtained by inserting IPC between the precedence related and expanded nodes.

Based on the above discussion, we are now ready to launch an overview of the total transition function — algorithm 10.

Algorithm 10 : *Our complete algorithm designed for making a transition from a current schedule into the next.*

4.3.3 Deadlock — A Possible Situation

Although it may not be immediately obvious, a most unwanted situation in terms of deadlock may occur during the application of our transition function. The example in figure 4.6 illustrates how this becomes possible.

In this example we see that the assignment/ordering comply with the “rules” of the transition function. Within each individual processor, the precedence constraints are maintained — \((n_4, n_1)\) as well as \((n_5, n_2)\) are independent node pairs, and \(n_3\) is a parent node to \(n_6\). Therefore, from “an isolated processor’s point of view”, this is a valid assignment/ordering.
However, it is impossible to arrange the nodes according to the global precedence relations. From the given assignment/ordering, the nodes \( n_3, n_4, \) and \( n_5 \) have to be executed first, but obviously they cannot be invoked prior to \( n_1 \) and \( n_2 \).

Now, in order to make the global arrangement and inserting IPC, assume that we begin selecting nodes from the top of "the schedule". For the particular node selected, we calculate its start time by taking into account 1) predecessors on other processors, and 2) the current status of the communication facilities. Using this strategy, the scheduling algorithm will converge if and only if as long as there are non-investigates nodes, it is possible to find at least one node for which it is true that

- it is the "next in line" and all its predecessors have been investigated (i.e., have been assigned a start time).

The term "next in line" means that the selected node is the next node (on that particular processor) — searching from top to bottom — which is non-investigated.

Turning back to the example in figure 4.6, we see that it is not at all possible to detect a node being both "next in line" and has all its predecessors already investigated. As a consequence, the schedule is invalid although the assignment/ordering (according to our definitions) is valid.

Note that the deadlock situation may not necessarily be detectable at the very beginning (as in the above example) of the scheduling process. For those DAPGs which have only one root node, this particular node is a parent node to every other node and thus according to our formulation of the transition rules, there will always be at least one processor which (in the initial phase of the scheduling process) has at least one schedulable node. If the assignment/ordering of such a graph leads to an invalid schedule, the deadlock situation therefore will not occur at the very beginning.

**How to avoid deadlock**

From our point of view there are at least two ways to work around the inherent deadlock situation.

First of all, we could redesign the strategy for slack calculation so that it becomes more restrictive. Basically, instead of relying only on parent and child nodes residing on the acceptor processor, also parent and child nodes residing on the other processors may influence the size of the slack. We have not developed any algorithms according to these ideas but it goes without saying that the overall search procedure would have an increased complexity as compared to the
procedure described above. Now, assume that we are very conservative in the sense that we take into account all parent and all child nodes on all processors. It means that deadlock would not be possible in any of the solutions generated by the transition function (all precedence relations are maintained by assignment/ordering). One could see this assignment/ordering as a schedule without IPC. The bad news about this strategy is that the search space would not equal the solutions space, i.e., $S_g \neq S_{total}$ (g being the overall scheduling function), and thus we would minimize the probability for finding the optimal solution.

The deadlock example from figure 4.6 can be slightly modified to illustrate our argumentation. In figure 4.7, the nodes $n_1$ and $n_4$ residing on processor $p_0$ have been swapped.

![Figure 4.7: Swapping $n_1$ and $n_4$ on processor 0, the assignment/ordering is still valid, but now it will not lead to a invalid schedule.](image)

Now, we can always find at least one node which 1) is “next in line” and 2) has all its predecessors investigated. The sequence in which the nodes will be investigated is \{n_1, n_5, n_2, n_4, n_6, n_3\} (one possible sequential schedule), and we conclude that there is no deadlock. This particular solution, however, (no matter if it turns out to have better or worse performance as compared to its predecessor solution) would be excluded from the search space in case we restrict the assignment/ordering to obey the global precedence relations. (n_2, n_4), (n_4, n_6), (n_2, n_3) and (n_3, n_5) are all node pairs which 1) belong to $S_g$ and 2) in the assignment/ordering are characterized by the same entry number — by definition, an invalid situation in case of global precedence maintenance.

Therefore, as a second suggestion for handling deadlock, we propose to preserve the strategy as devised in algorithm 10 knowing that it will lead to deadlock solutions. Anyhow, when a deadlock occurs, this particular solution is simply discarded and we generate an alternative one (of course from the same previous solution). This way of omitting deadlocks is also impacted by a certain overhead due to the waste caused by generating invalid solutions. What we gain, however, is that $S_g = S_{total}$ — our design criterion for the simulated annealing scheduler.

### 4.3.4 Employing the Metropolis Criterion

Once a valid schedule has been created, it is defined as “the next solution”, $j$. For this solution we must now calculate the cost in terms of makespan, as usually given by

$$m_j = \max_{n_a \in G} \{ s(n_a) + f(n_a) \}$$ ...

(4.8)

For the actual value of the control parameter, $c_k$, we apply $m_j$ as well as the makespan corresponding to the current solution, $m_i$, to calculate the Acceptance Probability for the next
solution.

\[ P_{ck}(\text{accept next}) = \begin{cases} 
1 & \text{if } m_j \leq m_i \\
\exp((m_i - m_j)/c_k) & \text{otherwise}
\end{cases} \]  \hfill (4.9)

In case of acceptance, the current solution is updated \((i := j)\). This includes the assignment/ordering as well as the makespan value.

### 4.3.5 Choosing an Appropriate Cooling Strategy

As we did mention in section 4.1.2, an appropriate cooling strategy is essential for the overall performance of the SA algorithm.

An example of a “single variable” cooling technique, \([46]\), is

\[ c_k = \alpha \cdot c_{k-1} \]  \hfill (4.10)

where \(\alpha\) is a scalar (normally \(0.80 < \alpha < 0.99\)), and \(k\) denotes the Markov chain number.

Due to a series of initial experiments, we found that this cooling strategy (for a fixed value of \(L\) which equal twice the number of nodes in \(G\)) is able to generate solutions which (in terms of makespan) are comparable to solutions obtained using a more sophisticated technique, \([44]\]. However, the number of iterations required to achieve a certain pre-specified makespan value may be reduced substantially, if we employ this other “multi-variable” strategy, equation 4.11. In general, we are therefore not better off in terms of makespan by using such an adaptive strategy, but we may reduce the overall run time. As a consequence, we thus have opted for this more complex technique — henceforth refered to as the cooling strategy.

\[ c_k = \frac{c_{k-1}}{1 + \frac{c_{k-1} \ln(1 + \delta)}{3\sigma_{c_{k-1}}}}; \quad k = 1, 2, \ldots \]  \hfill (4.11)

In this function, \(\delta\) is a distance parameter used to control the stationary step size — the larger the \(\delta\)-value, the larger the step size. \(\sigma_{c_{k-1}}\), on the other hand, represents the standard deviation of the makespans proposed in the \((k-1)^{th}\) Markov chain. For a given control parameter value, the next step size to be taken is therefore larger, the smaller the spreading.

**Some theoretical hints for the cooling strategy**

Here, it is out of the scope to discuss the complete set of theorems required to derive this strategy, but we will illustrate the most important basics behind it. As we have already seen, the annealing process is established by generating a sequence of solutions for every control parameter value. By proof it can be shown, that after a sufficiently large number of transitions at a fixed value of the control parameter \(c\), the SA algorithm will find a solution \(i \in S_{total}\) with a probability equal to

\[ P_c(X = i) \equiv q_i(c) \]  \hfill (4.12)

where \(q_i(c)\) is a probability distribution known in the SA literature as the equilibrium (or stationary) distribution.

Furthermore, it can be shown that if \(L_k\) denotes the length of the \(k^{th}\) Markov chain, then quasi equilibrium is achieved if \(a(L_k, c_k)\) — the probability distribution of the solutions after \(L_k\) iterations in the \(k^{th}\) Markov chain — is “sufficiently close” to \(q(c_k)\), i.e.,

\[ a(L_k, c_k) \approx q(c_k) \]
for a specified positive value of $\epsilon$.

Moreover, by an exhaustive proof, it can be justified that for equation 4.13 to hold for any $\epsilon$, the number of transitions required is quadratic in the size of the solution space (!). For any problem of even moderately size, this, of course, leads to $L$-values of non-practical sizes. Therefore, we need a less restrictive definition of quasi equilibrium.

Assume that the control parameter is increased infinitely ($c \to \infty$). Under this circumstance, all solutions are accepted, and thus

$$\lim_{c \to \infty} q(c) = \frac{1}{|S_{total}|}$$  \hspace{1cm} (4.14)

From this it follows that choosing sufficiently large $c_k$ values, quasi equilibrium is obtained because at these values all solutions occur with equal probability. However, since we have to decrease $c$, the uniform distribution of equation 4.14 cannot remain unaffected. Therefore, the length of the Markov chains and the decrement function for $c$ must be chosen such that quasi equilibrium is restored at the end of the individual Markov chains — in other words, we need to trade off the number of required control parameter steps and the Markov chain length. Large decrements in $c_k$ requires longer Markov chain lengths for quasi equilibrium to be restored at the next value of the control parameter, $c_{k+1}$.

A natural question to ask here is "how long should the Markov chain lengths actually be"? Since "next solutions" (i.e., transitions) are accepted with decreasing probability, we intuitively expect $L_k \to \infty$ for $c_k \downarrow 0$. This, of course, leads to an overall run time of non-practical use, and thus $L$ must be upper bounded to avoid extremely long Markov chains.

Now, for sufficiently small decrements of $c$, the stationary distribution of successive Markov chains are "close", and therefore we expect that quasi equilibrium is obtained after only a few transitions — given that quasi equilibrium was maintained at the previous Markov chain, i.e.,

$$\forall k \geq 0 : \| q(c_k) - q(c_{k+1}) \| < \epsilon$$  \hspace{1cm} (4.15)

for any positive value of $\epsilon$.

Taking this approach, our main concern now is to ensure quasi equilibrium at the very first Markov chain ($k = 0$). As discussed above, this can be implemented by selecting a sufficiently high $c_0$-value.

Equation 4.15 represents the fundamental requirement for the cooling strategy. However, in order to derive the strategy in full, substantial proofs need to be conducted. The interested reader is referred to [44] for these discussions. Henceforth, we simply employ the strategy as introduced in equation 4.11.

What only needs to be done is to discuss the number of transitions required in the Markov chains. [44] suggest in order for quasi equilibrium to be restored (using their cooling strategy), that for each Markov chain a number of transitions equal to the size of the neighbourhoods being proposed. We will investigate this figure for our SA scheduler.

### 4.3.6 Creating an Initial Assignment/Ordering

A question arises due to the initiation of the SA scheduler — how do we generate the very first random solution? Basically we must require that the solution is valid, i.e., the assignment/ordering must lead to a schedule without deadlock.
We therefore assign labels to the DAPG nodes according to their location in the critical path. Next, we select nodes from the top of the critical path, and assign the nodes successively to the processors which provides a valid order. Basically, this corresponds to the fundamental Hu scheduling algorithm as introduced in section 3.1.

Now, one may correctly argue that this is not a random assignment/ordering. This problem, however, finds its solution due to the fact that a $c_0$-value has to be calculated before the SA search is initiated.

Remember that the appropriate $c_0$-value corresponds to the control parameter value for which essentially all transitions are accepted. Now, assume that a sequence of $m_0$ (valid) transitions is generated at a particular $c$-value. From these $m_0$ transitions we count the number of cost decreasing and cost increasing transitions — denoted $m_1$ and $m_2$, respectively. For the $m_2$ cost increasing transitions, we calculate the average cost difference, $\Delta\text{cost}$ (as compared to the solutions from which they are generated). From equation 4.2 it must be true that

$$\chi \approx \frac{m_1 + m_2 \cdot \exp(-\frac{\Delta\text{cost}}{c})}{m_1 + m_2}$$

where $\chi$ is the acceptance ratio. From this equation we now isolate $c$,

$$c \approx \frac{\Delta\text{cost}}{\ln\left(\frac{m_2}{m_2 \chi - m_1 (1-\chi)}\right)}$$

The $c_0$-value can now be derived from equation 4.17 by generating the $m_0$ (valid) transitions — starting by the "Hu assignment/ordering" as introduced above. After each transition, $c$ is updated due to revised $m_1$, $m_2$, and $\Delta\text{cost}$-values.

Selecting $m_0$ sufficiently large, the "Hu assignment/ordering" is manipulated into a valid random initial solution for which we also through applications of equation 4.17 have the corresponding $c_0$-value.

### 4.3.7 The Overall SAS Strategy

According to the discussion so far, we are now able to set up an overall description of our SAS strategy. Algorithm 11 outline the major steps involved.

```
1) Label nodes in the DAPG according to the Critical Path;
2) Select nodes Highest Level First and assign to an available processor;
3) Calculate $c_0$ from a sequence of $m_0$ transitions (equ. 4.17) —
   this also provides an initial random assignment/ordering;
4) $k := 0$;
   While $c_k > c_{\text{stop}}$
     For $l = 1$ To $L$
       Repeat
         5) Generate new assignment/ordering and create schedule (alg. 10);
            Until valid schedule;
         6) Calculate makespan (equ. 4.8);
         7) Accept/Reject according to acceptance probability (equ. 4.9);
       EndFor;
     8) Calculate standard deviation for the $L$ makespans;
     9) $k := k + 1$;
     10) Calculate $c_k$ (equ. 4.11);
   EndWhile
```

**Algorithm 11:** An outline of the overall SAS algorithm.
4.4 Setting the SAS parameters

In order to tune the simulated annealing algorithm for our type of scheduling problem, we in this section describe how to obtain an initial setting of the parameters employed in the algorithm. Basically, we investigate

- the length, \( m_0 \), of the initial transition sequence
- the length, \( L \), of the Markov chains
- the distance parameter, \( \delta \)
- the stop criterion

Our purpose is to find a set of parameters which provides a trade-off between scheduling performance in terms of minimal makespan and fastest possible convergence.

Our means to obtain these values is to perform a set of scheduling experiment on a selected DAPG, and then adjust the individual parameters until a reasonable compromise between performance and run time is obtained. It can, of course, be argued that the set of parameters found using this approach is specific for the particular DAPG. Anyhow, we will apply these values as a first estimate for the set of parameters.

We have opted for a test DSP algorithm whose DAPG expose

- an extensive amount of inherent parallelism
- a complex but regular interconnection topology
- and a substantial sequential part

In the DSP community the algorithm is known as "The Roux-Gueguen Algorithm", [50], and it can efficiently be used to solve Toeplitz Systems on fixed point hardware. Algorithm 12 outlines the working of Roux-Gueguen's algorithm, and the corresponding atomic grain sized DAPG inheriting 72 nodes and 116 arcs is shown in figure 4.8 (note that only the computational part of the algorithm is shown — the initialization is assumed done elsewhere).

4.4.1 Length of the initial transition sequence

From equation 4.17 it is evident that the higher an acceptance ratio \( (0 < \chi < 1) \) we select, the higher a \( c_0 \)-value is required — this corresponds to the fundamental idea of "heating the material". It therefore seems reasonable to expect a longer initial "heating sequence", the higher we want the \( \chi \)-value.

In our test DAPG we now assume that all nodes have a runtime equal to one time unit and all arcs carry one token per sample period. Moreover, we assume that the target architecture is a fully connected message passing architecture with three processors and a transfer rate on the links equal to two time units per token.

For these constraints, we now investigate the evolution of \( c \) in 250 transitions for \( \chi \) equal to 0.8, 0.9, 0.95, and 0.99. The result is shown in figure 4.9.

From these observations we see very similar behaviour for the four different acceptance ratios. After approximately 100 transitions, it seems as though the \( c_0 \)-value settles at a level corresponding to the actual problem and the chosen \( \chi \)-value. For this test DAPG, we therefore conclude that 250 transitions are sufficient to find the appropriate start value for the control parameter.
Algorithm 12: The Roux-Gueguen algorithm basically has two parts — 1) initialization where the normalized autocorrelation coefficients \( r(i) \) are read into the "en" and "ep" data structures, and 2) the computation which provides the PARCO coefficients, \( k \).

What happens if we change the conditions for our scheduling problem? Let’s redo the experiments but now we want to schedule onto a six processor architecture and we also increase the communication rate to six time units per token. From figure 4.10 we basically observe the same kind of behaviour — after approximately 100 transitions, \( c_0 \) finds its level (which due to the increased problem complexity in all cases is somewhat higher as compared to the previous experiment).

Finally, we want to explore the \( c_0 \)-convergence in case the individual graph nodes have different execution times (we still assume all arcs to communicate one token per sample period). From a uniform distribution we assign random node weights in the interval \([4; 20]\) and then from 250 transitions we calculate \( c_0 \) for a six processor architecture which a six time unit per token communication rate. The results are shown in figure 4.11.

Again we observe the same overall picture — \( c_0 \) settles after approximately 100 transitions. Once more we recognize the increased \( c_0 \) levels due to the increased problem complexity.

From these experiments we therefore have strong reasons to believe that an initial transition sequence with a length equal to 250 is sufficient in order to find an appropriate start value for the control parameter.

From figures 4.9, 4.10, and 4.11 we also recognize that in most cases, the \( c \)-value is assigned a higher value (as compared to the value after 250 transitions) in the initial phase of the calculation. This may also be explained from equation 4.17. From this we see that \( c \) cannot be updated until the first cost increasing transition occurs. When it happens, only one value has contributed to \( \Delta \text{cost} \), and due to the initial critical path assignment/ordering (which actually is a reasonable solution), this value is likely to be fairly high. Due to the same argument, we may expect the first cost increasing transition to occur pretty soon (i.e., when it happens \( m_2 \approx m_1 \)), and thus (for a \( \chi \)-value close to one) the denominator will be close to zero.

Therefore, if in a specific situation it turns out that 250 iterations are not enough to settle the \( c_0 \)-value, we would probably start the search at a control parameter value which is higher than actually required for that particular problem. This, of course, influences the overall run time, but a reduced quality of the final solution is not to be expected.
Figure 4.8: The atomic sized DAPG for the computational part of a 6'th order Roux-Gueguen algorithm. The algorithm is composed of multiplication, division, addition, and “change sign” (CHS) operations.

4.4.2 Distance parameter and Markov chain length

As we have discussed already, we need to find a compromise between run time and performance. Essential to these overall characteristics are the distance parameter, $\delta$, and the length of the
Markov chains, $L$. Due to their close interaction, we select a-priori a value for one of them, and then tune the other.

According to section 4.3.5, the cooling strategy will work satisfactory with a Markov chain length equal to the size of the neighbourhood. For our transition function, we now argue that the neighbourhood size is upper bounded by $N^2$, where $N$ is the number of nodes in the DAPG. Staying at a particular solution $i$, we have $O(N)$ possibilities for selecting the next move node. Now, since the node has to be assigned to an arbitrarily selected new position due to a set of well defined slack conditions, this gives us at most $O(N)$ other possibilities — which makes up a total neighbourhood size of $O(N^2)$. In this argumentation we assume that one and only one IPC schedule exists for a particular assignment/ordering.

For a 100-node DAPG (which is not an unrealistic size), the neighbourhood size therefore would be upper bounded by 10000. Due to the deadlock problem, this overwhelming size is decreased somewhat, but still we estimate this figure to be of no practical use (at least for our experimental environment). We therefore take a different approach. As an initial guess, we make $L$ proportional to the size of the DAPG.

In order to provide a means for comparison, we schedule the test DAPG onto a three processor message passing architecture with a communication rate equal to two time units per token. We
basically employ the SAS strategy as described in algorithm 11 but the cooling is performed according to equation 4.10 with \( \alpha = 0.99 \). The acceptance ratio \( \chi \) equals 0.95 and the stop criterion is \( c = 0.005 \). We perform the experiment for Markov chain length equal to 100 and 200, respectively. Figure 4.12 shows the simulation results.

From these scheduling experiments it seems as though the set of near-optimal solutions has a makespan equal to 34–35 time units. For \( L = 100 \), a near-optimal solution is detected after approximately 700 control parameter steps (i.e., 70,000 transitions) whereas the near-optimal solution for \( L = 200 \) is found after 500 steps (i.e., 100,000 transitions). Since the transition function is basically random in its selection procedures, the detection of a particular solution is a stochastic variable with a certain probability. Therefore, we may obtain other results if we redo the experiments, but due to the high number of transitions proposed, it seems reasonable to assume that we will get something very similar. We therefore conclude that our initial selection of \( L = 100 \) is appropriate.

With the settle of \( L \), we now (for the cooling strategy in equation 4.11) make initial experiments with the distance parameter, \( \delta \), equal to 1.0, 0.75, 0.50 and 0.25. For \( L = 100 \), \( \chi = 0.95 \), and the stop criterion equal to \( c = 0.05 \) we again schedule the test DAPG unto the three processor message passing architecture with two time unit communication time. The simulation results
are shown in figure 4.13.

From these figures it seems reasonable to conclude that a distance parameter greater than 0.5 is out of the scope. For $\delta = 1.0$ and $\delta = 0.75$, the control parameter is decreased so rapidly that only very few cost increasing solutions ("up-hill" moves) are accepted. Despite the fact that SAS for these $\delta$-values finds a solution reasonable close to the near-optimal solution (before it reaches the $c_{stop}$-value), we may expect the SA algorithm to be potential for finding a quenched solution.

Before we draw any definite conclusion concerning the length of the Markov chains, we propose to redo the experiments above, i.e., for the four values of the distance parameter, we schedule the test DAPG onto the same architecture as before but now using Markov chains with twice the length, i.e., $L = 200$. One may argue that it will be without any effect because, as compared to the actual size of the neighbourhood it doesn't make sense to increase the length with only a factor of 2. On the contrary we argue that if it is necessary to increase the length far beyond, say 2-3 times the graph size in order to obtain reasonable solutions, the run time will prohibit any practical use of the strategy. The simulation results for $L = 200$ are shown in figure 4.14, where we note that the $x$-axis scaling has been changed as compared to figure 4.13.
Figure 4.12: In order to create a basis for comparison, we scheduled for two different Markov lengths the Roux-Gueguen algorithm employing the straightforward cooling strategy as proposed by Kirkpatrick. The x-axis show the number of control parameter steps taken before the stop criterion is reached.

Comparing these four scenarios we may still conclude that the control parameter decrease for \( \delta > 0.5 \) do not seem to allow acceptance of a substantial amount of cost increasing solutions. With the test DAPG as a reference, we therefore believe that a suitable \( \delta \)-value (in terms of performance/run time trade-off) belongs to the range \([0.25; 0.50]\). Regarding \( L \), it is interesting to observe the development in the search for \( \delta = 0.25 \). Several times we note an acceptance of a fairly high cost increasing solution — even when the SA algorithm has started its approach toward the near-optimal solution. These up-hill moves may be the reason for the good performance (makespan now down to 33) of this particular schedule. Although it may be due to the particular circumstances in the scheduling process, such up-hill moves were not observed for \( L = 100 \) (and \( \delta = 0.25 \)). However, in order to make any finite conclusion along that line, an exhaustive investigation of course is required, but as a guide-line we propose that the length of the Markov chains is chosen equal to at least the size of the DAPG.

4.4.3 The stop criterion

Up til now we have applied only one type of stop criterion — a fixed \( \epsilon \)-value. It is evident that this value strongly depends on the actual cooling strategy. According to the results shown in the figures 4.13 and 4.14, where \( c_{\text{stop}} = 0.05 \), this particular value seems as a reasonable choice. However, it definitely does not mean that it will suit all possible cases, since we have to maintain a reasonable performance/run time trade-off.

We note that a more optimal strategy (in terms of run time) may be due to the introduction of an adaptable stop criteria. For the example, if the makespan has not changed (at all, or e.g., to a lower value) for the last, say \( x \) steps of the control parameter, we may have some confidence
Figure 4.13: Makespan plotted against control parameter step for four values of the distance parameter using Markov chain length equal to 100.

Figure 4.14: Makespan plotted against control parameter step for four values of the distance parameter using Markov chain length equal to 200.

in that further optimization is fairly costly and thus we stop the search.

Similarly, we may calculate the variance of the makespan values (obtained on exit from the
Markov chains), again over say \( x \) steps of the control parameter. If the variance is lower than some prespecified value, then we stop. This strategy, however, must be carefully adjusted to suit 1) the particular scheduling problem, and 2) the cooling strategy — for very slow annealing, the makespan variance may become quite constant even in the early phase of the cooling, (and even if we sample over a substantial amount of control parameter steps). Therefore, this stop procedure may terminate the search too soon.

A third alternative is to select a fixed number of control parameter steps. In order for this strategy to work properly, we must know the number of steps required to reach the convergence region and thus there is an overhead associated with some initial experiments. The good news, of course, is that the run time is manifest.

4.5 Performance Evaluation of SAS

We are now ready to launch the performance analysis of our SAS strategy. The reader is referred to section 3.5 and appendix B for an exhaustive discussion on the random graph set created for these analysis. We just remind that the total set is composed as nine subsets which are divided into three categories characterized by graph sizes equal to 25, 50, and 100, respectively. Each category is further subdivided into sets having different degrees of graph parallelism, and each such set holds five DAPGs.

Due to the stochastic nature of SAS, the most true picture of its performance is probably found by scheduling the individual DAPGs several times and then let the individual speedups be represented by average values. Employing SAS in an experimental scheduling environment, the designer, however, is not likely to perform a number of SAS schedules of the same DAPG. Rather, he will use SAS only once in order to get some ideas of what is actually possible with the SA algorithm. Then, if it turns out that performance improvements seems likely as compared to e.g., list scheduling, then he may start experimenting with the SAS parameters and may eventually derive several schedules among which he next can choose the best.

In order to comply with this overall design behaviour, we have decided to apply SAS only once to each individual DAPG in a set. This provides, as was the case for our list scheduling experiments, five speed-ups from which we next calculate an average.

4.5.1 Choosing the SAS Parameters

According to the results of our initial SA scheduling experiments, we have chosen the following values for the SAS parameters.

- The acceptance ratio, \( \chi = 0.95 \)
- The initial transition sequence length, \( m_0 = 250 \)
- The distance parameter, \( \delta = 0.35 \)
- The Markov lengths, \( L = N \)
- The stop criterion becomes true when no performance changes have been observed for 100 successive Markov chains.

The stop criterion is chosen according to figures 4.13 and 4.14 from which it seems reasonable to terminate when no changes for at least 100 successive trials have occurred. For problems having a complexity comparable to our test vehicle, this criterion works nicely but it turns
out to be too restrictive for problems with a higher complexity, i.e., for graphs with 1) a high number of nodes having different runtimes, 2) a complex interconnection topology with different IPC times, and which are scheduled onto a high number of processors. In such cases the run time becomes unacceptably long because the convergence is much slower. In order to alleviate this problem, we extend the stop criterion formulation; the search terminates when no performance changes have been observed for 100 successive Markov chains or when max_{markov} Markov chains have been generated. Choosing max_{markov} = 500 we found in most cases that low complex problems terminates due to the “100 successively equal makespan” rule whereas for high complex problems, the termination is obtained according to the “max_{markov}” rule.

We believe that further tuning of the stop criterion in order to derive an even better performance/run time trade-off is certainly possible. Substantial experimentation with different types of graphs and architectures, however, is required in order to draw any definite conclusion about this matter.

### 4.5.2 Results of The SAS Experiments

We now present the results of scheduling the sets of randomly generated test graphs using SAS. We use the same procedure as described in section 3.5 where each graph set is scheduled onto 2-, 4-, and 6-processor message passing architectures with CCR equal to 0.5, 1, and 2, respectively. Table 4.1 shows the speedups derived by calculating the average speedup of the five individual DAPGs in each set.

In order to enable a discussion of the SAS performance, we next calculate the percentage difference in speedups between SAS and NODUST, and DLS, respectively. These figures are shown in tables 4.2 and 4.3.

These two tables show some interesting and somewhat identical results. It is evident that SAS is a superior alternative to NODUST/CP as well as to DLS, but it is also clear that SAS becomes more prominent the higher the CCR and the lower the number of processors.

For a low communication overhead, the speedup improvement never exceeds some few percents and in several cases we even see a performance degradation. For a high communication overhead, on the other hand, the improvement is obvious — with more than 50% in some cases. We explain
### Table 4.2: The percentage performance difference between NODUST/CP and SAS. Positive numbers indicate the percentage improvement of SAS over NODUST.

<table>
<thead>
<tr>
<th>Set #</th>
<th>$CCR = 0.5$</th>
<th>$CCR = 1$</th>
<th>$CCR = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=2</td>
<td>p=4</td>
<td>p=6</td>
</tr>
<tr>
<td>1</td>
<td>-1.26</td>
<td>-3.30</td>
<td>-3.75</td>
</tr>
<tr>
<td>2</td>
<td>0.10</td>
<td>2.21</td>
<td>-1.24</td>
</tr>
<tr>
<td>3</td>
<td>0.30</td>
<td>1.69</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>1.17</td>
<td>2.62</td>
<td>1.55</td>
</tr>
<tr>
<td>6</td>
<td>0.05</td>
<td>1.54</td>
<td>3.42</td>
</tr>
<tr>
<td>7</td>
<td>1.69</td>
<td>-0.81</td>
<td>-4.39</td>
</tr>
<tr>
<td>8</td>
<td>1.48</td>
<td>0.76</td>
<td>-3.10</td>
</tr>
<tr>
<td>9</td>
<td>1.37</td>
<td>1.72</td>
<td>-2.64</td>
</tr>
</tbody>
</table>

### Table 4.3: The percentage performance difference between DLS and SAS. Positive numbers indicate the percentage improvement of SAS over DLS.

<table>
<thead>
<tr>
<th>Set #</th>
<th>$CCR = 0.5$</th>
<th>$CCR = 1$</th>
<th>$CCR = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p=2</td>
<td>p=4</td>
<td>p=6</td>
</tr>
<tr>
<td>1</td>
<td>-3.30</td>
<td>-2.66</td>
<td>-3.12</td>
</tr>
<tr>
<td>2</td>
<td>-0.20</td>
<td>2.90</td>
<td>-1.83</td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>22.53</td>
<td>-0.04</td>
</tr>
<tr>
<td>4</td>
<td>0.37</td>
<td>-3.11</td>
<td>-3.24</td>
</tr>
<tr>
<td>5</td>
<td>0.45</td>
<td>1.42</td>
<td>0.18</td>
</tr>
<tr>
<td>6</td>
<td>0.25</td>
<td>1.10</td>
<td>2.37</td>
</tr>
<tr>
<td>7</td>
<td>0.37</td>
<td>-2.58</td>
<td>-4.88</td>
</tr>
<tr>
<td>8</td>
<td>0.86</td>
<td>-0.55</td>
<td>-5.90</td>
</tr>
<tr>
<td>9</td>
<td>0.15</td>
<td>0.57</td>
<td>-3.67</td>
</tr>
</tbody>
</table>
CHAPTER 4. STATIC SCHEDULING USING SIMULATED ANNEALING

this behaviour by the absence of the horizon effect in the SA technique. For a low communication overhead, we see that list scheduling strategies perform very well indeed, but due to the horizon effect, they are not superior in the presence of heavy IPC — in this situation a more stochastic search procedure, where several valid solutions are investigated, seems to perform substantially better.

Concerning the higher SAS-performance for a decreasing number of processors, we believe this has an obvious explanation. Increasing the number of processors, the complexity of the combinatorial optimization problem is also increased which means that in order to maintain the performance, more solutions have to be investigated, i.e., the Markov chains possibly have to be extended. Since we did not do that during the experiments, a performance decrease is to be expected when the number of processors is increased. The actual extension of the Markov chain length is still a subject for investigation.

Now, from the simulation results, other conclusions may be drawn. We observe that the occurrence of performance degradations (as well as very low improvements) is basically due to the same graph characteristic. The degradations are found in the sets 1, 4, and 7 (primarily for $p \geq 4$). These sets contain DAPGs with a low degree of parallelism, i.e., mainly sequential graphs. The result may therefore be explained by two circumstances. First of all we note that the list scheduling strategies employ the critical path for priority calculation which is obviously a very good strategy for sequential types of graphs. Secondly, we remember that the SAS technique 1) generates an initial solution by distributing the nodes uniformly onto the processors, and 2) select the acceptor processor from a uniform distribution. Due to these facts, scheduling a sequential DAPG in the presence of a high number of processors may therefore require more generated solutions before a good, say mono- or dual processor solution is found. In general we may conclude that the more sequential a DAPG, and the more processors present in the target architecture, the lower performance is found for maintained search complexity.

Concerning the graph sizes, there are no obvious distinctions. The only particular observation we want to point out is that for $N = 100$, there seems to be a slight overemphasis of performance degradations. Once again, this may be explained by a too limited search, and thus we expect this problem to vanish if we increase the Markov chain length.

4.6 Using SAS In the Presence of Heavy Communication

In section 3.7 we investigated the performance of NODUST/CP and DLS in the presence of a heavy communication overhead. The primary result was that NODUST/CP in most cases was able to outperform DLS, and NODUST/CP always found better solutions in the interesting cases where speedup is larger than 1.

Since we are also interested in the SAS performance when the IPC overhead is substantially larger than the computation, we now redo the experiments by using the SAS technique. Table 4.4 gives the NODUST/CP and SAS results in terms of speedups and the percentage performance improvement of SAS over NODUST/CP. For all SAS schedules we are using the same parameters as described above.

It is interesting to note that SAS in almost all cases is able to generate solutions with a speedup higher than 1. Only for sequential types of DAPGs, we observe parallel schedules with a lower performance as compared to the sequential solution. We may explain this by the actual working of SAS (as we also discussed above). Similarly, it is worth to point out that SAS outperforms NODUST/CP in all cases except for sequential DAPGs scheduled onto a high number of processors. In these cases, however, speedups are lower than or very close to 1, so they are of limited interest after all.
Table 4.4: The speedup measures for NODUST/CP and SAS in the presence of a very high communication overhead. Furthermore, the table gives the percentage performance improvement of SAS over NODUST/CP (i.e., negative numbers represent cases where NODUST/CP is outperforming SAS). The emphasized numbers indicate the important situations where SAS is generating speedups > 1.

Finally, we note that SAS for a low number of processors is able to generate solutions which in some cases are more than three time better as compared to those generated by NODUST/CP. This is an extension of the results shown in Table 4.2 where we in general also observe an increased performance for increased IPC overhead. In conclusion we state that the absence of the horizon effect in SAS becomes more and more important as the IPC overhead increases.
Chapter 5

Suggestions for SAS Redesign

So far we have discussed the fundamental of our simulated annealing scheduler. The core of the strategy is the transition function as described in section 4.3 which, through a number of stochastic selection procedures, enables evaluation of all solutions in $S_{total}$—an essential requirement for finding the optimal solution.

During the design of SAS, however, we came across the following question: Since we are allowing all possible solutions to be suggested and next evaluated with essentially equal probability, does it mean that we are visiting solutions which are far from optimal, and through which it doesn’t make sense to travel in order to arrive at the optimal solution? Immediately, the answer seems to be a “yes”. Remember for instance, that we in some cases are even generating invalid solutions in terms of dead lock.

Taking this discussion one step further, we may explain why SAS can get stuck in a near-optimal solution (this fact is easily demonstrated by scheduling the same DAPG a number of times, as we will see later in this chapter). Due to the stochastic nature of SAS, we are, of course, likely to find different solutions when scheduling a DAPG multiple times using identical SA parameters. According to equation 4.3, this problem may be alleviated by increasing the length of the Markov chains from $O(\alpha N)$ to, say $O(N^m)$, $O(\alpha^N)$, or even higher. Despite that we for an experimental scheduling environment are willing to accept relatively long compile times, everything higher than linear complexity seems to be out of scope.

In this chapter we therefore suggest some modifications to the basic SAS strategy which may guide the SAS algorithm more appropriately towards the final solution. Initially, we formulate a modification which maintains the computational complexity and next we suggest an alternative for compile time reduction.

5.1 Transition Functions with Guidance

According to the horizon effect which characterizes list schedulers we know that optimal local decisions may have non-optimal global consequences. From that perspective, the SAS approach is therefore appealing since it does not make any decisions (in terms of scheduling nodes) which, at a later point in time, may not be subject to a change. On the other hand, it does not make much sense to waste time assigning a node to the particular processor if this assignment is obviously leading to a low-performance solution and therefore has to be redone at a later point in time.

The question just is how to predict a priori if a certain transition— from a global perspective — will lead to a performance increase or decrease. Let’s try to come up with some suggestions.

In the presence of IPC, it seems reasonable to assume that assignments (of nodes to proces-
which minimize the necessary amount of communication is an attractive strategy. Our first approach is therefore to enforce communication dependent priority into the acceptor processor selection such that the (still randomly) selected move node \( n_j \) is assigned to an acceptor according to a precalculated processor selection probability distribution. We still have to stress the importance of \( S_g = S_{total} \) (\( g \) being the scheduling function) in order not to minimize the probability of finding the optimal solution. It is therefore required that all processors are potential for being selected as an acceptor, i.e.,

\[
P(\text{select } p_k) \neq 0, \quad k = 0..P - 1
\]  

(5.1)

where \( P \) is the number of processors in the architecture.

We design a probability distribution which is composed of 1) a basic uniform distribution among all processors, and 2) another distribution which reflects the partial communication into \( n_j \) from predecessors \( n_i \) residing on the \( P \) processors,

\[
P_{\text{basic}}(k) = \frac{1}{P} \quad k = 0..P - 1
\]  

(5.2)

\[
P_{\text{comm}}(k) = \frac{\sum_{\forall n_i \in g(p_k)} c_{ij}}{\sum_{\forall n_i} c_{ij}}
\]  

(5.3)

The final distribution is derived by normalizing and adding equations 5.2 and 5.3, i.e.,

\[
P(\text{select } p_k) = \gamma \cdot P_{\text{basic}}(k) + (1 - \gamma) \cdot P_{\text{comm}}(k) = \frac{\gamma}{P} + (1 - \gamma) \cdot \frac{\sum_{\forall n_i \in g(p_k)} c_{ij}}{\sum_{\forall n_i} c_{ij}}, \quad k = 0..P - 1
\]  

(5.4)

where \( \gamma \) is a parameter used to control the degree of stochastic selection.

For \( \gamma = 1 \), the processor selection distribution is uniform and thus the acceptor selection is totally stochastic — this corresponds to the plain SAS strategy. For \( \gamma \downarrow 0 \), the distribution is becoming more and more specified by the actual IPC pattern, i.e., the acceptor selection is being more and more guided.

Basically, \( n_j \) is guided towards assignment on the acceptor with which it has the highest communication overhead (from predecessor(s), \( n_i \), already assigned to the acceptor) as compared to the total communication into \( n_j \).

We note that in case the selected move node has no predecessors, the second term in equation 5.4 is undefined and thus the probability distribution is uniform.

For \( \gamma = 0 \), the distribution is totally determined by the IPC overhead from predecessors. This means that \( P(\text{select } p_k) = 0 \) if no predecessors are assigned to \( p_k \), and thus \( S_g \neq S_{total} \). Therefore, the following constraint is needed: \( 0 < \gamma \leq 1 \). We henceforth denote this scheduling function \( g \) “Simulated Annealing Scheduler with Guided Move”, SASGUM.

### 5.1.1 Optimizing In and Out Communication

One may argue that SASGUM has an inherent weakness in that it takes only into account communication into the move node. Therefore, SASGUM is providing a means to guide a selected node towards the processor with which it has the highest input communication. But in case a substantial part of successor nodes with high communication overhead is residing on other processors (different from the one which is given highest priority and therefore probably selected), IPC has to be implemented anyway — unless these successor nodes afterwards are moved to the same processor as their predecessor, i.e., the move node.
An alternative suggestion is therefore to measure both the communication into and out of the move node when the probability distribution for the acceptor selection is created. As stated above, the constraint \( S_g = S_{\text{total}} \) still has to be maintained and thus a basic (uniform) probability is required. We therefore employ equation 5.2 once more, and the communication dependent probability now becomes

\[
P_{\text{comm}}(k) = \frac{\sum_{\gamma_i \in g(p_k)} c_{ij} + \sum_{\gamma_i \in g(p_k)} c_{jl}}{\sum_{\gamma_i} c_{ij} + \sum_{\gamma_i} c_{jl}}
\]

(5.5)

where \( n_j \) denotes a successor to the selected move node \( n_j \).

The overall probability distribution is now created by normalizing and adding equations 5.2 and 5.5, which leads to

\[
P(\text{select } p_k) = \frac{\gamma}{P} + (1 - \gamma) \cdot \frac{\sum_{\gamma_i \in g(p_k)} c_{ij} + \sum_{\gamma_i \in g(p_k)} c_{jl}}{\sum_{\gamma_i} c_{ij} + \sum_{\gamma_i} c_{jl}}, \quad k = 0..P - 1
\]

(5.6)

where \( \gamma \) is the parameter (\( \gamma \in \{0; 1\} \)) for adjustment of the stochastic acceptor selection degree.

By the use of equation 5.6, the acceptor selection is guided by input communication to, as well as output communication from the move node. We therefore entitle this strategy “SASGUM_Io”.

Since we are assuming connected DAPGs, every node has at least one predecessor or one successor, and therefore the second term in equation 5.6 is never undefined.

Due to the communication minimization in both the forward and the backward direction, this technique tries to emphasize the collection or clustering of strongly sequentially dependent nodes on the same processor. All strategies discussed so far are (to a certain extend) doing that, but SASGUM_Io seeks to alleviate the horizon effect by “looking forward” into the schedule. Due to the lack of time, the SASGUM_Io performance evaluation unfortunately must be left for future work.

### 5.1.2 Performance Evaluation of SASGUM

In this section, we now present some results obtained using the SASGUM technique. As a first attempt, we like to investigate the performance in terms of achievable speedup. We schedule selected DAPGs from our sets of randomly generated graphs (see appendix B) onto fully connected message passing architectures with 2, 4, and 6 processors, using CCR equal to 0.5, 1, 2, and 10, respectively.

We have selected the sets 4, 5, and 6, which all include DAPGs with 50 nodes. The reason for selecting these sets is that we want to achieve a fair trade-off between graph size and required compile time. Scheduling a 50-node DAPG by our simulated annealing techniques using a SPARC10 workstation is done in 10-30 minutes, depending on parameters such as 1) the graph topology, and 2) the number of processors in the architecture. Increasing the graph size to 100 nodes, the compile time easily exceeds several hours (!). The sets 4, 5, and 6, are characterized by DAPGs having graph parallelism in the intervals [2;4], [6;8], and [10;12], respectively.

We recall that each set holds 5 DAPGs generated by using identical topology parameters. Therefore, in order to analyze the variations among different schedules of the same DAPG, we have chosen one graph from each set which is then scheduled five times. We selected graph number 1 from each set.

From the five speedup measures obtained, we calculate 1) the average, and 2) the spreading. The SASGUM experiments are all carried out for four values of the \( \gamma \)-parameter — 1.00, 0.75, 0.50, and 0.25 — the former corresponding to the pure SAS-algorithm (which we include for
\[ \gamma = 1.00 \]

<table>
<thead>
<tr>
<th>comm/proc</th>
<th>( \mu )</th>
<th>( \sigma )</th>
<th>( \mu )</th>
<th>( \sigma )</th>
<th>( \mu )</th>
<th>( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2/p2</td>
<td>1.905</td>
<td>4.34E-02</td>
<td>1.997</td>
<td>0.00E+00</td>
<td>1.998</td>
<td>7.04E-03</td>
</tr>
<tr>
<td>c2/p4</td>
<td>2.075</td>
<td>1.23E-02</td>
<td>3.812</td>
<td>4.01E-02</td>
<td>3.928</td>
<td>1.24E-02</td>
</tr>
<tr>
<td>c2/p6</td>
<td>2.049</td>
<td>4.33E-02</td>
<td>5.116</td>
<td>1.75E-01</td>
<td>5.756</td>
<td>1.21E-01</td>
</tr>
<tr>
<td>c4/p2</td>
<td>1.859</td>
<td>5.59E-02</td>
<td>1.986</td>
<td>8.10E-03</td>
<td>1.992</td>
<td>3.49E-03</td>
</tr>
<tr>
<td>c4/p4</td>
<td>1.957</td>
<td>6.40E-02</td>
<td>3.678</td>
<td>5.46E-02</td>
<td>3.945</td>
<td>3.04E-02</td>
</tr>
<tr>
<td>c4/p6</td>
<td>1.914</td>
<td>5.32E-02</td>
<td>4.691</td>
<td>1.35E-01</td>
<td>5.607</td>
<td>1.48E-01</td>
</tr>
<tr>
<td>c8/p2</td>
<td>1.593</td>
<td>8.43E-02</td>
<td>1.887</td>
<td>9.91E-02</td>
<td>1.987</td>
<td>5.68E-03</td>
</tr>
<tr>
<td>c8/p4</td>
<td>1.734</td>
<td>9.76E-02</td>
<td>3.448</td>
<td>9.37E-02</td>
<td>3.826</td>
<td>6.04E-02</td>
</tr>
<tr>
<td>c8/p6</td>
<td>1.720</td>
<td>4.11E-02</td>
<td>4.032</td>
<td>1.65E-01</td>
<td>4.962</td>
<td>1.92E-01</td>
</tr>
<tr>
<td>c40/p2</td>
<td>0.923</td>
<td>2.76E-01</td>
<td>1.214</td>
<td>1.74E-02</td>
<td>1.429</td>
<td>4.20E-02</td>
</tr>
<tr>
<td>c40/p4</td>
<td>0.746</td>
<td>1.15E-01</td>
<td>1.331</td>
<td>3.08E-01</td>
<td>1.560</td>
<td>1.04E-01</td>
</tr>
<tr>
<td>c40/p6</td>
<td>0.678</td>
<td>4.96E-02</td>
<td>1.409</td>
<td>1.69E-01</td>
<td>1.857</td>
<td>2.34E-01</td>
</tr>
</tbody>
</table>

Table 5.1: SASGUM simulation results for \( \gamma = 1.00 \) (i.e., SAS). All results for each degree of parallelism are derived from five independent schedules of the same DAPG.

The results are listed in the tables 5.1, 5.2, 5.3, and 5.4, respectively.

Figure 5.1: Speedup versus four values of the \( \gamma \) parameter for the graph from set 4 with a parallelism equal to 3. Graphs in the rows 1, 2, 3, and 4 have CCR equal to 0.5, 1.0, 2.0, and 10.0, respectively. Graphs in the columns 1, 2, and 3 have the number of processors equal to 2, 4, and 6, respectively.
### Table 5.2: SASGUM simulation results for $\gamma = 0.75$. All results for each degree of parallelism are derived from five independent schedules of the same DAPG.

<table>
<thead>
<tr>
<th>$\gamma = 0.75$</th>
<th>PAR=3</th>
<th>PAR=7</th>
<th>PAR=11</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm/proc</td>
<td>$\mu$</td>
<td>$\sigma$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>c2/p2</td>
<td>1.940</td>
<td>2.38E-02</td>
<td>1.996</td>
</tr>
<tr>
<td>c2/p4</td>
<td>2.079</td>
<td>4.37E-02</td>
<td>3.817</td>
</tr>
<tr>
<td>c2/p6</td>
<td>2.070</td>
<td>2.17E-02</td>
<td>5.236</td>
</tr>
<tr>
<td>c4/p2</td>
<td>1.869</td>
<td>3.46E-02</td>
<td>1.994</td>
</tr>
<tr>
<td>c4/p4</td>
<td>2.015</td>
<td>5.76E-02</td>
<td>3.678</td>
</tr>
<tr>
<td>c4/p6</td>
<td>1.959</td>
<td>8.86E-02</td>
<td>4.704</td>
</tr>
<tr>
<td>c8/p2</td>
<td>1.645</td>
<td>6.67E-02</td>
<td>1.914</td>
</tr>
<tr>
<td>c8/p4</td>
<td>1.860</td>
<td>4.78E-02</td>
<td>3.295</td>
</tr>
<tr>
<td>c8/p6</td>
<td>1.745</td>
<td>6.20E-02</td>
<td>3.903</td>
</tr>
<tr>
<td>c40/p2</td>
<td>0.910</td>
<td>1.58E-01</td>
<td>1.196</td>
</tr>
<tr>
<td>c40/p4</td>
<td>0.856</td>
<td>1.39E-01</td>
<td>1.227</td>
</tr>
<tr>
<td>c40/p6</td>
<td>0.818</td>
<td>1.49E-01</td>
<td>1.453</td>
</tr>
</tbody>
</table>

### Table 5.3: SASGUM simulation results for $\gamma = 0.50$. All results for each degree of parallelism are derived from five independent schedules of the same DAPG.

<table>
<thead>
<tr>
<th>$\gamma = 0.50$</th>
<th>PAR=3</th>
<th>PAR=7</th>
<th>PAR=11</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm/proc</td>
<td>$\mu$</td>
<td>$\sigma$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>c2/p2</td>
<td>1.952</td>
<td>5.96E-03</td>
<td>1.994</td>
</tr>
<tr>
<td>c2/p4</td>
<td>2.075</td>
<td>2.99E-02</td>
<td>3.861</td>
</tr>
<tr>
<td>c2/p6</td>
<td>2.067</td>
<td>2.62E-02</td>
<td>5.161</td>
</tr>
<tr>
<td>c4/p2</td>
<td>1.882</td>
<td>3.18E-02</td>
<td>1.997</td>
</tr>
<tr>
<td>c4/p4</td>
<td>2.016</td>
<td>5.83E-02</td>
<td>3.576</td>
</tr>
<tr>
<td>c4/p6</td>
<td>1.997</td>
<td>6.58E-02</td>
<td>4.800</td>
</tr>
<tr>
<td>c8/p2</td>
<td>1.716</td>
<td>5.79E-02</td>
<td>1.979</td>
</tr>
<tr>
<td>c8/p4</td>
<td>1.798</td>
<td>9.01E-02</td>
<td>3.350</td>
</tr>
<tr>
<td>c8/p6</td>
<td>1.760</td>
<td>1.32E-01</td>
<td>3.887</td>
</tr>
<tr>
<td>c40/p2</td>
<td>1.042</td>
<td>2.53E-01</td>
<td>1.194</td>
</tr>
<tr>
<td>c40/p4</td>
<td>0.843</td>
<td>9.23E-02</td>
<td>1.337</td>
</tr>
<tr>
<td>c40/p6</td>
<td>0.858</td>
<td>1.31E-01</td>
<td>1.396</td>
</tr>
</tbody>
</table>
\[ \gamma = 0.25 \]

<table>
<thead>
<tr>
<th>comm/proc</th>
<th>( \mu )</th>
<th>( \sigma )</th>
<th>( \mu )</th>
<th>( \sigma )</th>
<th>( \mu )</th>
<th>( \sigma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2/p2</td>
<td>1.928</td>
<td>1.37E-02</td>
<td>1.995</td>
<td>4.21E-03</td>
<td>1.992</td>
<td>3.49E-03</td>
</tr>
<tr>
<td>c2/p4</td>
<td>2.081</td>
<td>2.27E-02</td>
<td>3.755</td>
<td>6.70E-02</td>
<td>3.956</td>
<td>1.2E-02</td>
</tr>
<tr>
<td>c2/p6</td>
<td>2.070</td>
<td>2.17E-02</td>
<td>5.262</td>
<td>5.68E-02</td>
<td>5.853</td>
<td>1.28E-01</td>
</tr>
<tr>
<td>c4/p2</td>
<td>1.846</td>
<td>4.91E-02</td>
<td>1.991</td>
<td>3.87E-03</td>
<td>1.995</td>
<td>2.85E-03</td>
</tr>
<tr>
<td>c4/p4</td>
<td>2.027</td>
<td>4.28E-02</td>
<td>3.725</td>
<td>5.83E-02</td>
<td>3.956</td>
<td>3.07E-02</td>
</tr>
<tr>
<td>c4/p6</td>
<td>2.001</td>
<td>2.65E-02</td>
<td>4.723</td>
<td>9.62E-02</td>
<td>5.696</td>
<td>7.63E-02</td>
</tr>
<tr>
<td>c8/p2</td>
<td>1.660</td>
<td>6.99E-02</td>
<td>1.991</td>
<td>5.50E-03</td>
<td>1.991</td>
<td>1.13E-02</td>
</tr>
<tr>
<td>c8/p4</td>
<td>1.829</td>
<td>8.56E-02</td>
<td>3.354</td>
<td>1.46E-01</td>
<td>3.907</td>
<td>4.99E-02</td>
</tr>
<tr>
<td>c8/p6</td>
<td>1.822</td>
<td>6.89E-02</td>
<td>3.964</td>
<td>2.42E-01</td>
<td>4.949</td>
<td>3.42E-01</td>
</tr>
<tr>
<td>c40/p2</td>
<td>0.922</td>
<td>1.39E-01</td>
<td>1.233</td>
<td>3.94E-02</td>
<td>1.382</td>
<td>2.13E-01</td>
</tr>
<tr>
<td>c40/p4</td>
<td>0.821</td>
<td>1.37E-01</td>
<td>1.342</td>
<td>2.19E-01</td>
<td>1.630</td>
<td>3.05E-01</td>
</tr>
<tr>
<td>c40/p6</td>
<td>0.844</td>
<td>1.12E-01</td>
<td>1.493</td>
<td>1.78E-01</td>
<td>2.001</td>
<td>2.63E-01</td>
</tr>
</tbody>
</table>

Table 5.4: SASGUM simulation results for \( \gamma = 0.25 \). All results for each degree of parallelism are derived from five independent schedules of the same DAPG.

Figure 5.2: Speedup versus four values of the \( \gamma \) parameter for the graph from set 5 with a parallelism equal to 7. Graphs in the rows 1, 2, 3, and 4 have CCR equal to 0.5, 1.0, 2.0, and 10.0, respectively. Graphs in the columns 1, 2, and 3 have the number of processors equal to 2, 4, and 6, respectively.

From these results it is somewhat difficult to draw any firm conclusion concerning the makespan performance of SASGUM. A visualization, however, may help us detect some similarities. Figures 5.1, 5.2, and 5.3 shows the speedup versus the \( \gamma \)-values for the graphs with parallelism equal
Figure 5.3: Speedup versus four values of the $\gamma$ parameter for the graph from set 6 with a parallelism equal to 11. Graphs in the rows 1, 2, 3, and 4 have CCR equal to 0.5, 1.0, 2.0, and 10.0, respectively. Graphs in the columns 1, 2, and 3 have the number of processors equal to 2, 4, and 6, respectively.

... to 3, 7, and 11, respectively. Please note that these figures have been created by a graph drawing tool working in *autoscale mode* (otherwise some details would be hidden), which means that a *direct comparison of the individual graphs is impossible*.

From these graphs it seems as though SASGUM may be able to generate schedules which are comparable to or better than SAS (i.e., $\gamma = 1.0$). We base this statement on the fact that SASGUM in many cases shows the best performance. However, from these graphs alone, one has to be very careful drawing any definite conclusions because

- the results are average values implying that SAS for identical DAPGs and scheduling parameters may also outperform SASGUM.

- the spreading, which in most cases is non-zero, indicates that the SASGUM speedups are also varying. This indicates that SASGUM is also getting stuck into local minima.

- the variations between the average values (for the individual $\gamma$-values) within a single graph is in many cases so small (as compared to the actual speedup values) that the result may simply be due to the stochastic nature of the strategies, and therefore cannot express any specific behaviour.

- in most cases there is no obvious identity between the shape of the graphs for the same number of processor or communication ratio.

There is, however, one particular indication which is very clear. For a *low degree of parallelism* (i.e., figure 5.1), we see in *all cases* an improvement of SASGUM over SAS for *all values* of $\gamma$ (the...
exception being for $\gamma = 0.25$, 2 processors and CCR=1.0). This observation may be explained by the following arguments. When we are scheduling an essentially sequential DAPG, we have a high number of strictly connected nodes which are likely to have no connections to other nodes. Therefore, for such a DAPG it is less complicated to cluster sequentially dependent nodes as compared to more parallel DAPGs where we may expect the presence of more nodes having multiple in- and output connections. Since SASGUM is designed with clustering of such strongly dependent nodes in mind, this characteristic is reflected in its superior performance for a low degree of parallelism.

Although there also seems to be some identities between the graphs representing the same number of processors (e.g., in figure 5.1, the optimal $\gamma$-value turns out to be 0.5 for a low number of processors), we hardly can draw any conclusion from these observations because the characteristics may change if we choose e.g., other DAPG topologies but having identical parallelism. Therefore, from the experiments conducted so far, our most firm conclusion concerning the speedup performance is that SASGUM seems suitable for mainly sequential types of DAPGs. However, further experimentations with more 1) DAPGs, 2) $\gamma$-values, and 3) CCR-ratios are required in order to finalize a solid conclusion on the SASGUM-performance.

Finally, we want to give a few comments on the spreading derived from the multiple scheduling of identical DAPGs.

- For a low degree of parallelism, the spreading is fairly constant for varying number of processors. This indicates that SASGUM (and SAS) for sequential graphs is scheduling somewhat independent of the architectural size. This can also be easily demonstrated by throwing more processors into the architecture — if the DAPG do not have the sufficient parallelism in order to utilize these processors efficiently, they are not included in the final schedule.

- For a higher degree of parallelism we see in many cases that the spreading increases the more processors available. This indicates that when 1) the DAPG has sufficient parallelism in order to utilize the processors, and 2) the solution space is expanded, the more variation is to be expected in the final solutions. It may mean that the number of solutions proposed and evaluated may not be sufficient. Increasing the length of the Markov chains or decreasing the distance parameter used in the cooling strategy are possible adjustments which may solve this problem.

- We observe in most cases an increased spreading for ultra high communication overheads as compared to the corresponding cases with a more moderate communication. The reason may be that the very high communication overhead is making the optimal solution "less obvious" because communication scheduling is now becoming more important, i.e., more solutions have to be investigated.

In conclusion, these spreading measurements have shown us the importance of proper parameter adjustment due to variations in the DAPG and architectural characteristics. As expected, we found a need for more exhaustive searches the more complex our problems are. Still, however, we point out the always annoying compile time increase which must be accepted due to a better search performance.

### 5.2 Combining List and SA Scheduling

So far we have proposed SAS modifications which in some cases may lead to improvements in the final solutions. From our conclusion, however, we still have to face the problem of finding a
CHAPTER 5. SUGGESTIONS FOR SAS REDESIGN

fair trade-off between the required performance and an acceptable compile time. In this section we suggest a method for lowering the compile time while maintaining the performance.

In section 4.3.6 we devised a method for initiating the SAS search. A critical path based assignment and ordering scheme is employed in order to set up the initial solution. This solution is then “randomized” by shuffling selected move nodes around until the initial control parameter value \( c_0 \) has settled. Then the actual SA search is launched.

Using this familiar method, we now perform a typical SAS-scheduling of a 100-node DAPG from our test set onto a 6-processor architecture with \( CCR = 2 \). The acceptance ratio and the initial \( c_0 \) sequence equal 0.95 and 250, respectively, the distance parameter is 0.5, the lengths of the Markov chains are 100, and the stop criterion is either 100 successive equal makespans or at most 500 generated Markov chains.

If we take a look at the convergence characteristic, shown in figure 5.4, we realize for the actual setting of the SAS parameters that approximately 20% (i.e., 100) of the suggested solutions are used to get within 20% of the final makespan (as compared to the makespan of the initial solution). This is a very reasonable overhead for an SA algorithm, but it may worse substantially if we go for a slower annealing by decreasing the distance parameter.

![A Typical SAS Convergence Curve](image)

Figure 5.4: Choosing arbitrarily a 100-node DAPG from our test set and schedule it onto a 6-processor architecture using SAS, we get this typical convergence characteristic.

From this observation we conclude that it may be possible to obtain a faster convergence if the search is launched from a near-optimal solution instead of being initiated from a “more or less” random assignment/ordering. The question now is how to obtain a near-optimal solution? Obviously, it has to be done essentially free of charge — otherwise we wouldn’t gain anything. According to 1) the results published by Liao et al. [43], and 2) our investigations described so far, an immediate suggestion is to employ the Dynamic Level Scheduling heuristic. As compared to
the compile time overhead characterizing SAS, the DLS computational complexity is negligible. The DLS algorithm therefore seems as a good tool for generation of an initial near-optimal solution which next by simulated annealing is further optimized. This has motivated us to propose a hybrid List SA scheduler (to be called LISA hereafter) which 1) generates an initial near-optimal solution using the DLS list scheduling heuristic, and then 2) uses one of our SA techniques (SAS or SASGUM) to converge towards the optimal solution.

In the SAS technique we are "heating" the solution until it becomes sufficiently random and simultaneously we obtain an initial control parameter value. Assuming that we want to start the search from the near-optimal solution found by DLS, no heating should take place (otherwise we would simply destroy the solution). Instead, we have to determine the control parameter value \( c_0 \) which corresponds to the near-optimal solution. Our method for tuning \( c_0 \) to a specific solution is therefore discussed next.

5.2.1 Matching \( c \) to a Near-Optimal Solution

Previously we employed the acceptance ratio, \( \chi \), for the \( c_0 \)-determination and we generated new solutions until the specified ratio was obtained. We are now in a somewhat different situation because the initial solution \( s_{\text{init}} \) is already fixed. It is therefore a matter of increasing \( c_0 \) until it has a value which corresponds to \( s_{\text{init}} \). Algorithm 13 outlines our technique.

```
set the Increase Acceptance Ratio (IAR %);
\( s_{\text{init}} \) represents the initial DLS solution;
\( c_0 \approx 0 \);
\( \Delta_{\text{accept}} = 0 \);
While \( \Delta_{\text{accept}} < \text{IAR} \) do
  {\n    \( \Delta_{\text{accept}} = 0 \);
    For \( l = 1 \) to 100
      {\n        generate \( s_j \) from \( s_{\text{init}} \);
        if cost(\( s_j \)) > cost(\( s_{\text{init}} \)) then
          if exp((cost(\( s_{\text{init}} \)) - cost(\( s_j \)))/\( c_0 \)) > random [0,1] then \( \Delta_{\text{accept}} = +1 \);
      }
    increase \( c_0 \);
  }
```

Algorithm 13: Our algorithm for \( c_0 \)-determination given an initial near-optimal DLS solution.

The essential idea is to reflect what is going on during a search. This is done by generating a sequence of new solutions, \( s_j \), from \( s_{\text{init}} \). If such a new solution increases the cost (as compared to \( s_{\text{init}} \)), the acceptance probability (equation 4.9) is calculated and next compared to a random number derived from a uniform distribution. If the new solution is "accepted", a counter \( \Delta_{\text{accept}} \) is increased. At the end of the sequence, the value of this counter is compared to an Increase Acceptance Ratio (IAR) which specify the percentage of accepted cost increasing solutions to be allowed. To match \( c_0 \) properly to the near-optimal solution, IAR must be assigned a small value, e.g., 1%. We keep incrementing \( c_0 \) until \( \Delta_{\text{accept}} \geq \text{IAR} \).

It is evident that the computational complexity of this function must be substantially smaller than the overall computational complexity required by SAS to derive from the initial solution to a near-optimal solution comparable to the DLS solution.

The sequence of proposed solutions has a length which matches the length of a Markov chain
CHAPTER 5. SUGGESTIONS FOR SAS REDESIGN

for a 100-node DAPG (this is convenient due to the percent specification of IAR, but, of course, it can easily be changed). Therefore, in order for the \( c_0 \)-calculation to have considerable lower computational complexity, the number of times \( c_0 \) is increased must be far smaller than the number of c-steps required to reach the DLS solution from a random initial solution. This rises the question on how the \( c_0 \)-increasing function should work. Some suggestions are reported below.

**Increasing the initial control parameter**

The most obvious and simple procedures for increasing \( c_0 \) are

\[
e_0 = \alpha \cdot \text{counter} + \beta; \quad \alpha > 0, \quad \beta > 0, \quad \text{counter} \geq 0
\]

and

\[
e_0 = \alpha \cdot e_0; \quad \alpha > 1
\]

In both cases, the problem is how to select proper values for the constants. We opt initially for the latter method, since here only one parameter has to be found. \( \alpha \) must be chosen in order to make trade-off between

- as few iterations as possible
- as low a \( c_0 \)-value as possible

The former requirement asks for a high \( \alpha \)-value in order to rapidly reach a suitable \( c_0 \), whereas the latter demands a low \( \alpha \)-value in order to ensure small step sizes (so that a too high \( c_0 \)-value is avoided).

Let’s give some examples. Assume first of all that we want to schedule our Roux-Gueguen test graph from figure 4.8. In a first situation, 1) all nodes have the same runtimes (equal to 1), and all communications are uniform and equal to 2. Furthermore, we schedule onto 3 processors, IAR equals 2\%, the initial \( c_0 \)-value is chosen to 0.01, and \( \alpha \) is 1.5. Table 5.5 shows for 10 independent runs the percentage acceptance rates due to increasing \( c_0 \)-values. For the actual choice of parameters, we get the clear indication that \( c_0 \) is found after approximately 10-11 iterations. Now, if we change 1) the node runtimes to be uniformly distributed in the interval [4;20], 2) all communications to take 8 time units, and 3) increase the number of processors to 6 (and all other parameters are remained unchanged), we find that \( c_0 \) is obtained after 12-13 iterations, i.e., only a slight increase in the computational overhead.

If we, as a second example, want to schedule a much more irregular DAPG, say a 100-node graph from our library of randomly generated graphs (e.g., graph 1, set 8), we get similar results. First we choose uniform execution- and communication times for all nodes and arcs, respectively, and we schedule onto 3 processors. Given the same LISA-parameters as above, we obtain \( c_0 \) in about 10 iterations. Selecting the run times from a uniform distribution of numbers in [4;20], increasing the communication rate, and changing the number of processors to 6, \( c_0 \) is again found in 12-13 iterations.

This almost similar behaviour for different DAPGs, of course, is a nice feature, but a reasonable explanation for this facility may be the exponential increase of \( c_0 \). Table 5.5 illustrates somewhat the phenomenon — in 5 out of 10 cases, the percentage of accepted cost increasing solutions is in one step becoming at least twice as high as the specified IAR-value. This indicates that the step size is suddenly too large and thus leading to a too high final \( c_0 \)-value.
Table 5.5: The percentage of accepted cost increasing solutions as a function of increasing $c_0$. The numbers in brackets indicate the final $c_0$-values.

In order to work around this problem we are looking for a more flexible (i.e., adaptive) algorithm for $c_0$-determination. One suggestion may be to increase $c_0$ using an $\alpha_1$-factor until the first cost increasing solution(s) is observed. Then (after a possibly small decrease of $c_0$), the $c_0$-increase is continued using another factor, $\alpha_2$, where $1 < \alpha_2 < \alpha_1$. Figure 5.5 shows an example of such a multi-factor function where $\alpha_1 = 1.5$ and $\alpha_2 = 1.1$. The first accepted cost increasing solution is assumed being detected at the 9th iteration and after the factor-exchange is performed, the $c_0$-increase is continued.

Figure 5.5: Increasing $c_0$ may be done by a multi-factor function in order to find a $c_0$-value which is not too large.

For illustration purposes, we redo the $c_0$-determination experiment (IAR=2%, and $c_{0,\text{init}} =$
Table 5.6: 10 independent runs showing the number of accepted cost increasing solutions versus the number of iterations — bracket numbers indicate the final c_0-values.

0.01) for the Roux-Gueguen test graph (uniform execution and communication) using the multivariable c_0 increase function from figure 5.5. The results from 10 independent runs are shown in table 5.6.

As expected, we see that the average number of iterations required is increased somewhat. However, we now obtain final c_0-values which correspond much better to the actual DLS solution given a prespecified IAR-value (in this case 2%). In 7 out of 10 possible situations, the algorithm terminates at the right IAR-value, and in only one case, the number of accepted cost increasing solutions is twice as high as specified. Beside, we get a substantial reduction in the final c_0-value. On the average, we are now down to 0.551 as compared to 0.711 in the previous experiment where only one increase factor was employed. This has a significant influence on the number of iterations required in the following SAS procedure.

Further optimizations may be possible by careful tuning of e.g., the increase factors. Alternatively, more than two factors may smooth the c_0-determination even better. Also the initial c_0-value may be adjusted in order to bring down the number of iterations — so far we have observed no acceptance of cost increasing solutions before the 8th iteration, although, of course, it is possible to appear, due to DAPG variations and architectural parameters.

For the rest of the thesis, the two-factor c_0-increase function (using the values described above) will be employed whenever we are referring to LISA.

As an example of the working of LISA we schedule the Roux-Gueguen test graph (figure 4.8), and we next compare the result obtained to the result from a pure SAS scheduling. We are assuming uniform node execution and communication times, and the message passing architecture has 3 processors. For both LISA and SAS we are using 1) Markov chains with 72 trials (corresponding to the number of nodes in the Roux-Gueguen DAPG), 2) a distance parameter equal to 0.35, and 3) the stop criterion is either 100 identical makespan values or at most 400 Markov chains. For SAS we are using 250 iterations to settle c_0 at an acceptance ratio χ of 0.95, and for LISA, IAR equals 2%. The makespan evolutions are plotted against the number of iterations in figure 5.6.

We note that the SA part of LISA is using the pure SAS strategy which, of course, may be substituted by SAGUM if wanted. From figure 5.6 we see that LISA is basically working as expected. The initial DLS solution ensures a significant decrease in the makespan upon entry to
the SA part — in this example by a factor of 2. We see that DLS, also as expected, produces a near optimal solution (this is concluded from the fact that our SAS algorithm is actually able to further optimize the schedule). The makespan is decreased from 46 to 38 before the stop criteria is activated (i.e., a makespan reduction of 17%). As compared to the pure SAS technique we recognize that the reduction in initial makespan/control-parameter values leads to an earlier activation of the stop criteria. In the example, the number of iterations used, is down with more than 20%. However, as we might expect (because of the identical SA techniques), LISA do not outperform SAS in terms of the final makespan.

We may conclude, however, that LISA is potential to find better solutions as compared to SAS if we in both strategies employ a stop criterion which terminates the search after a fixed number of iterations. In such a case, LISA is able to spend more trials close to the optimal solution, increasing the probability that this particular solution will be detected. If, on the other hand, the run time is a critical issue, LISA may be used to terminate the search at an earlier stage as compared to SAS by applying a stop criterion which uses the makespan values to decide termination.

From the example, it also seems as though our algorithm for $c_0$-determination is working properly. The $c_0$-value which is found for the DLS solution is leading to a controlled and continuously decrease in makespan although small cost increases — which we know are essential for the escape from local optimum — are allowed.

In order to draw any definite conclusion on LISA, a more thoroughly investigation is required, but onwards we simply apply the technique as it has been introduced here.
Chapter 6

Case Studies

In the previous chapters, we have based our investigations on randomly generated test DAPGs, and from these experiments we have gained substantial knowledge of the scheduling strategies' behavior and performance.

In order to further broaden the scope of our work we now want to demonstrate the workings of the strategies when applied to a set of "real-world" DAPGs. In the ideal case, we would like to use a benchmark library of DSP algorithms. Unfortunately, due to the absence of substantial research material in the field of DSP multiprocessor scheduling, we have not be able to find any paper, text book, or technical report describing such a library.

In the High-Level Synthesis community, researchers defined in the early days one very popular benchmark example for comparison of their scheduling algorithms — the 5th order Digital Wave Filter, [13]. However, based upon this example we cannot make fair comparisons with these earlier works — basically because the HLS scheduling strategies are assigning atomic operations onto a heterogeneous target architecture composed of execution units (EXU) like ALUs, multipliers, shifters, etc. This puts some special constraints on how many different types of operations that may be executed within the individual control steps.

An obvious possibility is therefore to create our own DSP DAPG benchmark library. There is, however, one severe limitation which prevent us from doing that exhaustively. Within the frames of this thesis we have no immediate access to a parser which can transform an executable specification in terms of C, C++, Silage, or whatever, into an equivalent SDF notation as required for input to our strategies. Since in this project it is out of scope to design such a tool, the only alternative is to hand craft a number of DSP SDFGs/DAPGs (like the Roux-Gueguen algorithm employed previously). As we will discuss below, this has been done for a few well-known DSP algorithms.

Another problem related to the absence of a text-to-graph compiler is how we detect the optimal grain sizes of the DAPGs. As discussed in chapter 1, the grain size problem is crucial for optimal scheduling. For the DSP algorithms, however, we have decided to use the smallest possible grain size, i.e., the algorithms are expressed in terms of atomic operations. We are aware of the fact that this may not be optimal, basically, due to three reasons:

- From an IPC point of view, the MIN/MAX problem is presumably not optimally solved if we opt for the atomic grain size. It may mean that better schedules for the individual algorithms are likely to exist given larger granularities.

- From an architectural point of view, the atomic grain size is in some cases a very disadvantageously choice. The reason being that most state-of-the-art PDSPs have address
CHAPTER 6. CASE STUDIES

generation units which are optimized for certain types of algorithms, e.g., 1) FIR filters for which the delay line can be updated with almost zero overhead in terms of circular buffers, and 2) FFT types of algorithms for which bit-reversed addressing modes are normally supported. It means that distribution of such algorithms onto multiple processors (at least from the atomic grain size point of view) may prevent these efficient architectural facilities from being efficiently employed.

- Another advantageous architectural property of most PDSPs which cannot be applied is their support of zero overhead loop handling. In all our cases we are therefore completely unfolding the loops which is believed to prohibit optimal utilization of the architectures.

We accept, however, these penalties since our primary goal is to make qualitative comparisons of our scheduling strategies. But, on the other hand, it should not go without saying that DSP applications, when normally presented as block diagrams, in many cases consist of a few (say, less than 20) cascaded functional units. From such large grain sized block based algorithmic descriptions, it is quite easy to recognize if utilization of parallelism is possible. If not, it is definitely required to look inside these blocks to see if the individual algorithms do have some inherent parallelism that can be efficiently exploited. Therefore, we certainly have confidence in the approach we are taking towards the atomic/small grain size description of the algorithms.

Another question that we have to take a stand on is the option for inclusion of data-dependent algorithms. Since we previously have assumed only data-independent algorithms (and in case of data-dependent computation have discussed the necessity for applying worse case execution times), we therefore limit ourselves to investigations of only data-independent algorithms.

As a consequence, we have decided to derive the DAPGs, and next make scheduling experiments for the following DSP algorithms.

- The 6th order Roux-Gueguen algorithm
- The 4th order Durbin algorithm
- The 4th order Cholesky factorization
- The block based Autocorrelation
- The recursive Autocorrelation
- The 16-point DIT FFT
- The 5th order Wave Digital Filter

Apart from the Roux-Gueguen algorithm, which was presented in section 4.4, these algorithms are now introduced below.

6.1 Selected Off-the-Shelf DSP Algorithms

The purpose of this section is to give a brief overview of the DSP algorithms selected for the scheduling case studies. Basically, they are all "off-the-shelf" algorithms, and thus we do not describe in detail their theoretical background, but rather give a pseudo code presentation, and next show their DAPGs. We note, however, that a large class of advanced signal processing problems are related to "estimation". Indeed least-square estimation is frequently used for such objective as parameter estimation and direction finding. The most computational intensive step
involved is often the solution of a set of linear equation, \( Ax = b \). We therefore, among others, investigate a number of algorithms which may be employed under various circumstances in order to generate, manipulate, and solve such a matrix equation.

### 6.1.1 The Durbin Algorithm

As we did explain in section 4.4, the Roux-Gueguen algorithm is efficiently employed for fixed-point solution of Toeplitz Systems, i.e., a set of linear equations where (when expressed in matrix form) the matrix is symmetric and the elements along any diagonal are identical.

Prior to the algorithm design by Roux & Gueguen, Durbin contributed a similar algorithm for Toeplitz solving which, however, did not have the same nice fixed-point facilities. On the other hand, the Durbin algorithm do not need a succeeding coefficient transformer in order to provide the solution of the equation system (which in some cases is required by Roux-Gueguen). Durbin’s algorithm is designed to solve Toeplitz systems where the column vector on the right hand side comprises the same elements as found in the matrix, [51]. Algorithm 14 outlines the Durbin algorithm.

```
# INITIALIZATION
E_0 = r(0);

# COMputation
FOR m = 1 TO p
    k(m) = \frac{-r(m) + \sum_{i=1}^{m-1} a_{m-1,i} \cdot r(m-i)}{a_{m,m}};
    a_{m,m} = k(m);
    FOR i = 1 TO m - 1
        a_{m,i} = a_{m-1,i} + k(m) \cdot a_{m-1,m-i};
    ENDFOR;
    E_m = E_{m-1} \cdot (1 - k_m^2);
ENDFOR;
```

**Algorithm 14**: Pseudo code for the \( p \)th order Durbin algorithm. We assume that the autocorrelations coefficients, \( r(m) \), which are the known variable in the equation system, have been calculated elsewhere. \( a_{i,j} \) denotes the unknown variable (for all systems with order 0 to \( p \), \( E \) is the so-called residue variance, and \( k \) represents the PARCO coefficients (same as in the Roux-Gueguen algorithm).

In order to manage the manual creation of a precedence graph for the Durbin algorithm, we have opted for \( p = 4 \). The resulting DAPG, which has 37 nodes and 56 arcs, is shown in figure 6.1. Note that arithmetic operations supplied with precalculated operands (i.e., the \( r \)-coefficients) and constants, do not have their inputs specifically shown on the graph.

As compared to the Roux-Gueguen algorithm, we observe that the Durbin algorithm has 1) a much more irregular interconnection topology, and 2) less inherent parallelism. Further, for simplicity, the “sign change” operation required for the calculation of \( k \) is now assumed to be included in the division operation.

In order to execute the Roux-Gueguen algorithm as well as the Durbin algorithm, we need to calculate the reciprocal. We note that some PDSPs have division primitives included in their instruction set, while others (typically older types) do not have support for division. In the former case, the number of clock cycles required for the calculation of a division instruction (for a certain word length) is used as the estimated execution time, while in the latter situation, a
division macro is inserted. It means that in both cases, we are assuming the division to be an indivisible operation which has to be executed sequentially on a single processor.

### 6.1.2 The Cholesky Factorization

The Roux-Gueguen and Durbin algorithms discussed so far are designed in order to solve Toeplitz equation systems. In many cases, however, the matrix is not Toeplitz but only symmetrical. In such situations, a method for matrix factorization (into upper and lower triangular matrices which are next suitable for forward/backward substitution), known as the Cholesky factorization [52], may be efficiently employed. The algorithm outline is shown in algorithm 15.

Once more we have opted for \( p = 4 \) which leads to the DAPG illustrated in figure 6.2. The graph, which is mainly characterized by two sequential (but not independent) parts, has 30 nodes and 40 arcs.

In the Cholesky factorization, we see that several square root operations are required. This
Algorithm 15: Pseudo code representation of the Cholesky factorization algorithm. As usual, $p$ denotes the order, i.e., the dimension of the matrix. $L_{ij}$ is the $(ij)^{th}$ element in the lower triangular matrix, and $A_{ij}$ represents the elements in the matrix to be decomposed.

![Algorithm Diagram](image)

Figure 6.2: The DAPG for the $4^{th}$ order Cholesky factorization.

type of operation is normally not included in PDSP instruction sets. Only the most recent processors have support for square root computation in terms of a reciprocal square root seed which is next used by an iterative algorithm in order to derive at the required accuracy. The higher accuracy needed, the more iterations. The designer is responsible for the creation of this algorithm. It means that the square root computation has to be performed by a small function/procedure which we assume indivisible and for which we have to estimate the execution time dependent, of course, on the accuracy required. On most processors for which this seed function is not provided, a complete square root estimator has to be software implemented.

We finally note that in case the matrix is non-symmetric, the Cholesky algorithm must be substituted by a more general LU decomposition like e.g., Gauss elimination. Apart from the
square root, the standard algorithms for LU factorization are (in terms of their computational nature) quite similar to Cholesky, and thus we do not discuss these types of matrix decomposition.

6.1.3 The Block Based Autocorrelation

In many types of DSP applications, the autocorrelation of the incoming signal has to be calculated. Basically, the autocorrelation calculation is a means to create a substantial data reduction while maintaining the informational content of the signal, [53]. The fundamental block based autocorrelation algorithm, which assumes that a block of input samples are available, is shown in algorithm 16.

![Algorithm 16: Pseudo code representation of the standard block based autocorrelation algorithm.](image)

Figure 6.3: The direct transformation of the textural autocorrelation specification provides this characteristic “sum-of-product” precedence graph.

![Figure 6.4: Rearranging the summation order, this tree structured DAPG can be derived.](image)

From the algorithm we note the following assumptions. Normally, each block (or frame) of the incoming signal, $s[n]$, is multiplied by a weighting function, a “window”, prior to the
autocorrelation calculation itself. This, however, is assumed to be done elsewhere. Secondly, due to this windowing, the displacement of the two sequences \( s[n] \) and \( s[n+l] \) caused by increasing \( l \)-values may lead to a computation reduction caused by zero multiplications. We do not take this feature into account here. Finally, we note from algorithm 16 that the calculation can be performed 1) one autocorrelation coefficient at a time, or 2) successively update of all coefficients simultaneously. This is basically a matter of interchanging the two loops. Choosing the one over the other may have some impact on the memory access scheme. However, since we are mainly interested in the DAPG topology, we have opted for the "coefficient-by-coefficient" method as described by the algorithm. In this case, it is evident that no dependences exist between individual coefficient calculations, and therefore these may be derived in parallel. Consequently, we analyze the computational aspects of calculating one coefficient, which is a typical sum of products computation. This may be performed in terms of a directly transformation of the algorithm into the DAPG as shown in figure 6.3 where we have chosen \( M = 32 \). Alternatively, the calculation can be done by using the tree topology as illustrated in figure 6.4. We note that both DAPGs have identical computational complexities (32 MUL and 31 ADD) which makes them suitable for comparison in terms of different interconnection topologies.

6.1.4 The Recursive Autocorrelation

From the previous section it is evident that autocorrelation calculations may have a high computational complexity. Since the autocorrelation functionality is required in a broad range of applications, researchers have devised various algorithms which try to minimize this computational overhead. One of these suggestions is the recursive form, which efficiently updates all autocorrelation coefficients on a sample-by-sample basis, [54].

![Diagram of the non-atomic recursive autocorrelation estimator.](image)

Figure 6.5: The synchronous data flow graph of the non-atomic recursive autocorrelation estimator, and the atomic view of the internal filter structure. Empty nodes are fork nodes.

Here, it is out of scope to derive all the theory behind this algorithm, but rather we give a brief introduction and present the SDF and the DAP graphs, figures 6.5 and 6.6, respectively. The basic idea is to make use of the fact that the windowed autocorrelation can be thought of as the convolution of the input signal and the window sequence;
Figure 6.6: Maintaining the granularity as proposed by the SDFG, the precedence graph exposes (in this case) 13 parallel executable filters (F).

\[
R_l(m) = \sum_{n=-\infty}^{\infty} s[n] \cdot w[m-n] \cdot s[n-l] \cdot w[m-n+l]
\]

(6.1)

where \( R_l(m) \) denotes the \( l^{th} \) autocorrelation coefficient (or lag) for the window displacement \( m \).

Instead of choosing a finite length window (as we did assume above), a window which has an infinite length (but being small outside the analysis interval) is now applied. Such a window can be realized in terms of the impulse response of a second order filter with two real poles.

Next, using the complex convolution theorem, it can be shown that the Z-transform of the window product (equation 6.1) is a transfer function of a third order recursive filter with a set of filter coefficients being specific for the actual lag. Figure 6.5 shows the autocorrelation coefficients as output from such filters being exposed by delayed and multiplied samples of the input signal. The internal filter structure is shown as well.

In our scheduling experiments we assume that the individual filters are executed sequentially on one processor (i.e., the internal parallelism is not exploited), and thus the filters are each represented by a single node in the precedence graph. We note, however, that since we are assuming connected graphs, a dummy node with zero execution time has been added. This can be done without any loss of generality. Following our assumption of “data available” (see appendix A), the precedence links from this node may be considered as having zero communication overhead. From this it follows, that the recursive autocorrelation — in this example for order \( p = 12 \) — is a set of parallel executable digital filters. This invites to optimal scheduling of this DAPG with 27 nodes and 26 arcs.

### 6.1.5 The Fast Fourier Transform

For signal analysis purposes, the Fast Fourier Transform (FFT) has been one of the most popular algorithms over the years. Strictly speaking, the FFT is not a Fourier transform in the strong sense, but rather an efficient calculation scheme of the well-known discrete Fourier transform [53],

\[
F(k) = \sum_{n=0}^{N-1} f[n] \cdot e^{-j2\pi kn} \quad (6.2)
\]

Since the FFT has been exhaustively discussed in numerous text books, it makes no sense to do it also here. Instead, we derive the DAPG for a 16-point Decimation In Time (DIT) FFT, which we have chosen for inclusion in our case studies, figure 6.7. In this graph, which has 96 nodes and 136 arcs, it should be noted that all arithmetic operations are complex. Therefore, they
may be divided into atomic (real) operations which, however, is not considered here. Similarly, we have to take into account that the communications on all arcs are complex numbers.

6.1.6 The Wave Digital Filter

We mentioned previously that researchers have used the 5th order wave digital filter as a benchmark example for scheduling experimentation, [13]. Since this example has a long standing tradition, we also include it here.

Wave Digital filters (WDF) are modeled after classical analog filters in lattice or ladder config-
urations. They have excellent properties concerning coefficient accuracy requirements, dynamic range, and especially all aspects of stability when implemented in terms of fixed-point arithmetic. The interested reader may consult [55] for a detailed review of this class of digital filters. Here, on the other hand, we omit the algorithmic specification but present instead 1) the SDFG of a 5th order WDF, figure 6.8, and next 2) the corresponding DAPG, figure 6.9. The filter contains 26 additions and 8 filter coefficient multiplications.

### 6.2 An ADSP-2106x Based Target Architecture

In the following, we demonstrate the scheduling of our selected DSP algorithms onto a homogeneous message passing PDSP architecture. We refer to section 2.5 for a detailed description of the underlying IPC model being used.

We have opted for the Analog Devices ADSP2106x SHARC multiprocessor PDSP, [23, 56], as our target processor. This processor has been chosen for our experimental studies basically because it is among the most recent PDSPs developed with multiprocessing in mind. Below, we list some of its features.

- The 2106x features six 4-bit link ports.
CHAPTER 6. CASE STUDIES

- These ports can be clocked twice per cycle, allowing each to transfer 8 bit per cycle.
- Each port can be operated independently and simultaneously.
- Link port data is packed into 32 or 48 bit words, and can be directly accessed by the core processor or DMA-transferred to on-chip memory.
- Each link port has its own buffered input and output registers.
- Handshaking controls the direction of communication
- The DMA controller allows zero-overhead data transfer without core processor intervention.
- The DMA controller operates independently of and invisibly to the core processor, allowing DMA operations to occur while the core is simultaneously executing computational instructions.
- The ADSP-2106x is code- and function-compatible with the ADSP-2102x processor.

The ADSP-21000 family core has an instruction set with several multifunction instructions. This enable the execution of computations in parallel with data transfers between on/off-chip memory and the register file used for 1) load of operands, and 2) saving of results by the independent execution units (i.e., the multiplier, the ALU, and the shifter). We also note that the architecture (and the instruction set, of course) allow an ADD and a MUL operation to be executed in parallel. Unfortunately, since our scheduling strategies cannot generate schedules which have multiple sub-algorithms executing at the same time (at the same processor), see equation 2.11, we cannot utilize this facility. We can, however, utilize the feature of simultaneous execution and data move. Every instruction can be executed in a single processor cycle.

In our scheduling experiments we assume all calculations to be performed using 32-bit floating point precision. Based on this assumption, we now from [56] derive the number of clock cycles required by the operations found in our set of benchmark DSP algorithm, table 6.1. We distinguish between three types of DAPG nodes,

- External, i.e., nodes which get all their operands from the external world, e.g., from an ADC. In the DAPGs, such nodes are the root nodes.

- Internal, i.e., nodes which get some of their operands from other nodes in the DAPG. Such nodes have at least one input arc.

- Exclusive Internal, i.e., nodes which get all their operands from other nodes in the DAPG.

The reason for this classification is due to a difference in the number of clock cycles — caused by the requirement for additional load instruction in some cases. The example shown in figure 6.10 illustrate the difference between the three node types.

We assume that the operands are stores in memory. In order to execute the external node, two operations which load the operands from memory into the register file are therefore needed. Due to the multiple data busses and memories (data and program) found on the ADSP-2106x, such two memory-to-register transfers are executable in a single cycle. On termination, the result from an arithmetic operation is automatically written to a register in the register file (specified by the instruction), and thus (since the next instruction reads its operands from the register file) the internal and the exclusive internal nodes requires one and zero heading load
Figure 6.10: External, internal, and exclusive internal nodes are characterized by a difference in the number of load instructions required. Here we illustrate the load overhead for real atomic arithmetic operations.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>ADD/SUB</th>
<th>MUL</th>
<th>DIV</th>
<th>SQRT</th>
<th>FILT</th>
<th>ADD/SUB$^*$</th>
<th>MUL$^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERNAL</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>18</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>18</td>
<td>-</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>EXCL INT</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>17</td>
<td>10</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.1: The number of clock cycles required by different types of arithmetic operations when executing on a ADSP-2106x as either external, internal, or exclusive internal nodes. Overheads for initialization purposes, e.g., load of filter coefficients into the register file, are not present in these numbers, since they need to be executed only once. Addition and multiplication marked with (*) denotes complex operations. Empty fields represent numbers not required for our experiments.

instruction, respectively. Note that non-atomic nodes, e.g., complex operations, may need more load instructions.

Figure 6.10 also indicates the possible IPC transfers which become active if connected nodes are scheduled on different processors. On the message passing ADSP-2106x architecture, we assume the processors to be interconnected through the dedicated communication links. Since 1) the ADSP-2106x CPU can directly access the link port registers (like any other universal register in the architecture), and 2) the amount of data to be transferred between the DAPG nodes is limited to 1-2 words per arc, we describe the communication behavior as being without the need for DMA support. It means basically that we assume

- intra-processor communication to be performed with zero overhead through the register file
- IPC to be 1) initiated as write of partial results to the link port registers on the transmitting processor and 2) finalized as read of partial results from the link port registers on the receiving processor.

According to these assumptions and the message passing communication model illustrated in figure 2.12, the overall IPC consist of 1) a data move instruction (1 cycle) for each word to be
transmitted (moving data from the register file to the link port register), 2) a setup time required
for the transmitting processor to gain the link, 3) the actual link transmission time, and 4) a
data move instruction (1 cycle) for each word to be received.

Now, since our scheduling strategies do not schedule a new communication at a point in
time when the required link is already busy, we safely assume that link contention will occur
so infrequently that the setup time can be ignored. Consequently, the number of clock cycles
required for IPC is

\[ IPC \text{ clock cycles} = 6 \cdot \# \text{ data words} \tag{6.3} \]

because 2 cycles are necessary for write/read, and 4 cycles are needed for the actual transfer of
a 32-bit word.

It has to be pointed out, that the 2 write/read instructions actually are executed on the CPU
and therefore correctly ought to be counted in together with the numbers given in table 6.1.
However, it would mean variable node clock cycle counts as to whether connected nodes are
scheduled on the same or on different processors — additional write/read instructions would
not be necessary for intra-processor communications. In the current versions, our scheduling
strategies cannot handle the situation of variable node execution times, and therefore we decided
to include the overhead in the transfer time.

As compared to the figures for the computational time of atomic operations given in table 6.1,
we realize that the communication overhead is relatively high.

### 6.3 The Scheduling Results

In order to prepare for scheduling, we first of all annotate the DAPG nodes with the estimated
node execution times found above (table 6.1). This allows us to calculate the inherent degree of
parallelism of each graph (equation B.1). The results of this analysis are shown in table 6.2.

As expected, we see that the graphs have quite some differences in the degree of parallelism.
Somewhat surprising, though, is the rather low parallelism of the Roux-Gueguen algorithm.
On the other hand, it is easily explained in terms of the compute intensive critical path (all
division operations are residing here). Another interesting observation is the substantial increase
in parallelism we may gain by reorganizing the sum-of-product autocorrelation calculation from
a direct (and thus very sequential) to a tree structured graph. The high degree of parallelism
found in the FFT and the recursive autocorrelation was expected.

Based on these results, we get some hints (note that the communicational aspects are not
included here) on the number of processors which may be efficiently utilized in a parallel architec-
ture. The sequential nature of the Durbin and the Cholesky algorithms seem to restrict their
need to no more than 2-3 processors. The Roux-Gueguen algorithm, the Wave Digital Filter, and
the directly implemented autocorrelation may be utilizing efficiently as much as 3-4 processors,
while the remaining algorithms seem to have sufficient inherent parallelism in order to exploit an even higher number of processors.

Besides, due to the six dedicated communication links provided on the ADSP-2106x, we may realistically assume fully connected target architectures with 2-6 processors. A reasonable choice is therefore to perform our scheduling experiment on 2, 4, and 6 processors, respectively.

Our benchmark algorithms are all scheduled using Hu', NODUST/CP, DLS, SAS, SASGUM, and LISA (SAS). For the SAS strategy, the following parameters are employed. 250 iterations are used in order to find the initial random solution, applying an acceptance ratio (\(\chi\)) equal to 0.95. The distance parameter (\(\delta\)) is in all cases 0.35, and the stop criterion is either 100 successive identical or at most 400 proposed solutions. From the lesson learned in section 4.5.2 concerning the expected performance improvements as related to the number of processors, we have opted for Markov chains with lengths equal to \(2N\), although for some DAPGs it may be an overkill (and probably a too low value for others). This same set of parameters is also used for SASGUM which, additionally, requires the \(\gamma\)-value (ratio between stochastic and deterministic acceptor selection). In these case studies we have opted for 0.5, knowing from the experiments conducted in chapter 5 that this value may not be optimal for all DAPGs. Nevertheless, we see this choice as a reasonable alternative as a first approach.

For SAS and SASGUM, we perform each scheduling five times in order to alleviate somewhat the spreading problem. From these five results, we extract the best and the worst speedup achieved.

Since the main purpose of LISA is to optimize the DLS solution with a "reasonable" run time overhead, we take this as an argument to perform only one LISA scheduling of each benchmark algorithm (should we derive five schedules, the intention of lowering the overhead would be somewhat nonsense). Like any other result generated by our SA based schedulers, the LISA results are samples of a stochastic process and thus another run may provide better or worst solutions. However, we want to show 1) the performance of this algorithm as compared to DLS (in terms of speedup), and 2) its computational complexity as compared to SAS. Consequently, we have chosen the IAR-factor as low as possible (i.e., 1\%) in order to initiate the search as close as possible to the near-optimal DLS solution (this decision may mean that in some cases the initial c-value is too small for the SAS algorithm to escape efficiently from the local optima). Secondly, since the LISA search is launched close to the optimal solution, it is reasonable to believe that the convergence curve has already flattened, and thus we choose the stop criterion to be either 50 successive identical or at most 400 proposed solutions. It may mean a saving in the number of created Markov chains, at the risk for a too early termination. The speedup results are reported in tables 6.3, 6.4, and 6.5.

In the following we now present some comments on the scheduling results. First of all, we see that the degree of inherent parallelism characterizing the individual algorithms has the expected effect on the speedups. Examples are the sequential Durbin and Cholesky algorithms which for all scheduling strategies show only limited performance improvements as the number of processors is increased. The recursive autocorrelation, on the other hand, has sufficient parallelism (and a limited IPC overhead) to utilize an increase in the number of processors. The FFT also has a high degree of parallelism but further, it is characterized by many potential IPC sources which (at least for the SA based strategies) seem to exclude "the almost" linear speedup (as seen for the recursive autocorrelation).

If we take a closer look at the heuristics (i.e., Hu', NODUST, and DLS), we recognize the following essential results.

- For the very sequential algorithms, the heuristics provide the same performance for a low
### Table 6.3: The speedups derived when scheduling onto 2 processors. A_TR and RE_A denote the tree structured and the recursive autocorrelations, respectively. The double entry in the SAS and SASGUM strategies corresponds to best/worst result in five schedules.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>ROUX</th>
<th>DURB</th>
<th>CHOL</th>
<th>AUTO</th>
<th>A_TR</th>
<th>RE_A</th>
<th>FFT</th>
<th>WDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hu*</td>
<td>0.860</td>
<td>1.013</td>
<td>1.274</td>
<td>0.519</td>
<td>0.766</td>
<td>1.733</td>
<td>0.802</td>
<td>0.946</td>
</tr>
<tr>
<td>NODUST/CP</td>
<td>0.860</td>
<td>1.013</td>
<td>1.342</td>
<td>1.000</td>
<td>0.950</td>
<td>1.733</td>
<td>0.906</td>
<td>1.082</td>
</tr>
<tr>
<td>DLS</td>
<td>1.084</td>
<td>1.013</td>
<td>1.319</td>
<td>0.556</td>
<td>0.833</td>
<td>1.733</td>
<td>0.976</td>
<td>1.060</td>
</tr>
<tr>
<td>SAS</td>
<td>1.206</td>
<td>0.961</td>
<td>1.285</td>
<td>1.301</td>
<td>1.793</td>
<td>2.000</td>
<td>1.556</td>
<td>1.472</td>
</tr>
<tr>
<td>SASGUM</td>
<td>1.220</td>
<td>1.000</td>
<td>1.312</td>
<td>1.301</td>
<td>1.860</td>
<td>2.000</td>
<td>1.618</td>
<td>1.325</td>
</tr>
<tr>
<td>LISA</td>
<td>1.229</td>
<td>1.026</td>
<td>1.355</td>
<td>1.357</td>
<td>1.827</td>
<td>2.000</td>
<td>1.400</td>
<td>1.152</td>
</tr>
</tbody>
</table>

### Table 6.4: The speedups derived when scheduling onto 4 processors. A_TR and RE_A denote the tree structured and the recursive autocorrelations, respectively. The double entry in the SAS and SASGUM strategies corresponds to best/worst result in five schedules.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>ROUX</th>
<th>DURB</th>
<th>CHOL</th>
<th>AUTO</th>
<th>A_TR</th>
<th>RE_A</th>
<th>FFT</th>
<th>WDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hu*</td>
<td>1.217</td>
<td>1.026</td>
<td>1.367</td>
<td>1.377</td>
<td>1.610</td>
<td>3.391</td>
<td>1.986</td>
<td>1.359</td>
</tr>
<tr>
<td>NODUST/CP</td>
<td>1.433</td>
<td>1.026</td>
<td>1.367</td>
<td>1.462</td>
<td>2.021</td>
<td>3.391</td>
<td>1.892</td>
<td>1.082</td>
</tr>
<tr>
<td>DLS</td>
<td>1.316</td>
<td>1.026</td>
<td>1.367</td>
<td>1.462</td>
<td>1.795</td>
<td>3.391</td>
<td>2.222</td>
<td>1.293</td>
</tr>
<tr>
<td>SAS</td>
<td>1.229</td>
<td>0.940</td>
<td>1.367</td>
<td>1.863</td>
<td>2.794</td>
<td>3.714</td>
<td>2.037</td>
<td>1.472</td>
</tr>
<tr>
<td>SASGUM</td>
<td>1.322</td>
<td>0.987</td>
<td>1.367</td>
<td>1.793</td>
<td>2.794</td>
<td>3.714</td>
<td>2.979</td>
<td>1.472</td>
</tr>
<tr>
<td>LISA</td>
<td>1.402</td>
<td>0.963</td>
<td>1.367</td>
<td>1.939</td>
<td>2.794</td>
<td>3.714</td>
<td>3.011</td>
<td>1.395</td>
</tr>
</tbody>
</table>

### Table 6.5: The speedups derived when scheduling onto 6 processors. A_TR and RE_A denote the tree structured and the recursive autocorrelations, respectively. The double entry in the SAS and SASGUM strategies corresponds to best/worst result in five schedules.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>ROUX</th>
<th>DURB</th>
<th>CHOL</th>
<th>AUTO</th>
<th>A_TR</th>
<th>RE_A</th>
<th>FFT</th>
<th>WDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hu*</td>
<td>1.654</td>
<td>1.026</td>
<td>1.367</td>
<td>2.111</td>
<td>2.500</td>
<td>4.588</td>
<td>3.011</td>
<td>1.359</td>
</tr>
<tr>
<td>NODUST/CP</td>
<td>1.675</td>
<td>1.026</td>
<td>1.367</td>
<td>2.262</td>
<td>2.436</td>
<td>4.588</td>
<td>3.333</td>
<td>1.082</td>
</tr>
<tr>
<td>DLS</td>
<td>1.720</td>
<td>1.026</td>
<td>1.367</td>
<td>2.209</td>
<td>2.436</td>
<td>4.588</td>
<td>3.333</td>
<td>1.293</td>
</tr>
<tr>
<td>SAS</td>
<td>1.338</td>
<td>0.940</td>
<td>1.367</td>
<td>2.005</td>
<td>3.276</td>
<td>4.875</td>
<td>3.943</td>
<td>1.472</td>
</tr>
<tr>
<td>SASGUM</td>
<td>1.521</td>
<td>0.987</td>
<td>1.367</td>
<td>2.317</td>
<td>3.276</td>
<td>4.875</td>
<td>3.315</td>
<td>1.472</td>
</tr>
<tr>
<td>LISA</td>
<td>1.697</td>
<td>1.026</td>
<td>1.367</td>
<td>2.209</td>
<td>3.167</td>
<td>4.875</td>
<td>3.218</td>
<td>1.472</td>
</tr>
</tbody>
</table>
as well as for a high number of processors (the exception being Cholesky scheduled onto 2 processors where NODUST is superior). This means essentially that all heuristics are not employing more processors than necessary despite they are available in the architecture.

- For 1) the Roux-Gueguen algorithm, 2) the direct autocorrelation, and 3) the wave digital filter (which all have a parallelism of 3) no immediate conclusion can be drawn. It seems as though, however, that NODUST and DLS are doing better than Hu* (the exception being the wave digital filter on 4 and 6 processors) leading to the confirmation that 1) duplication and 2) dynamic levels are means to optimize the schedule for DAPG with a non-trivial interconnection topology. DLS is never outperforming NODUST on the autocorrelation which may mean that duplication is suitable for algorithms characterized by a long sequential part along which many nodes are connected. (We note in parenthesis that none of the heuristics are even close to the optimal partitioning of the autocorrelation).

- The recursive autocorrelation which has a very limited IPC overhead (due to many independent subalgorithms) is scheduled equally well (but not optimally) by all heuristics. This emphasizes our knowledge concerning Hu types of strategies as being near-optimal in the absence of IPC.

- The DLS strategy is doing equally well or better than Hu* and NODUST for the FFT algorithm on all architectures. This again is an indication of the reasonable Dynamic Levels technique for complex interconnection topologies.

In a while, we will show some measures of the required CPU time for the individual strategies. These figures basically emphasize our previous statement of substantial compile time increases when we select one of the SA based strategies. If one is willing to accept this penalty, table 6.3, 6.4, and 6.5 show some interesting results from which we extract the following observations:

- In most cases, the SA based strategies are able to find equivalent or better solutions as compared to the solutions derived by the list based heuristics. The few exceptions are all observed for 4 and 6 processors. This mainly indicates that the length of the Markov chains and/or the distance parameter need adjustment for a high number of processors.

- For 2 processors, the worst schedules generated by SAS are in most cases better than the schedules derived by the list scheduling techniques (the exceptions being the Durbin and the Cholesky algorithms). The best SAS schedules are always superior to the list schedules. For 4 processors, we see in 5 out of 8 cases that the worst SAS schedule is equivalent to or better than the list schedule, and only in one situation (the Durbin algorithm), the best SAS schedule has lower speedup as compared to the list heuristics. In the case of 6 processors, half of the worst SAS schedules have a lower performance as compared to the list strategies which is only true in 3 situations for the best SAS schedules. The bad performance for 6 processors is observed for the Roux-Gueguen algorithm, the Durbin algorithm, and the FFT. These are all algorithms which have a somewhat complex interconnection topology, again indicating the need for an adjustment of the essential SAS parameters for problems with large solution spaces.

- Comparing the best schedules generated by SAS and SASGUM, we observe the following. For a 2-processor architecture, SASGUM finds in 6 of 8 cases equivalent or better solutions. Increasing the number of processors to 4, we see that in only one situation SAS is superior to SASGUM. The same is true for 6 processors. This indicates that SASGUM, in particular when the solution space is enlarged due to more processors, is performing well.
Table 6.6: This table shows the number of CPU seconds consumed during the scheduling of our benchmark algorithms using a SPARC10 workstation.

- For 2 processors, LISA is always further improving the DLS solutions. For 4 processors, LISA is in only one case degrading the initial DLS solution, while this happens twice for 6 processors. This may mean that for a high number of processors, 1) the SAS parameter need adjustment, 2) the selected 50/400 stop criterion should be reconsidered, or 3) the initial $\alpha^0$ value is too high, suggesting a reformulation of equation 5.8 to take into account the number of processors (i.e., $\alpha$ being a function of $P$).

- Comparing the LISA schedules to the best SASGUM schedules, we find that for 2 processors, LISA is beaten in 5 of 8 cases. For 4 and 6 processors, on the other hand, LISA generates equivalent or better solutions in 5 of 8 situations. This is consistent with the observation that for a low (i.e., 2) number of processors there are in most cases a significant performance deviation between the DLS and the best SASGUM solutions. This is not true for a high (i.e., 6) number of processors, meaning that for 6-processor architectures LISA is using its iterations close to the optimal solution which does not seems to be the case for an architecture with a small number of processors.

- We find in all cases that LISA is outperformed on the tree structured autocorrelation. This corresponds nicely to the fact that for this particular algorithm, the DLS heuristic is never superior. This means that LISA, in cases where DLS (as compared to the other list heuristics) do not generate the best solution, may require 1) a higher initial $\alpha^0$ value in order to escape efficiently from the DLS solution, or 2) an extended length of the Markov chains. These indications are also supported by the sequential Durbin and Cholesky algorithms for which LISA is always (the exception being for Durbin on 4 processors) finding solutions equivalent to the best solution derived by SAS/SASGUM. In these cases, DLS generates solutions which are only slightly improved by SAS or SASGUM, and thus we may conclude that the DLS schedule is close to optimal.

For illustration of the computational complexity, we next show some figures of the CPU resources required by the various strategies in order to schedule the benchmark algorithms onto the 4-processor architecture using a SPARC10 workstation, Table 6.6. For SAS and SASGUM, the CPU times reported correspond to the scheduling experiments in Table 6.4 which provided the best solutions (i.e., the highest speedups).

One should note that the results derived for the SA based strategies are also samples of a stochastic process. This is basically because the search may get trapped into a local optimum and thus the "50 or 100 successive identical solutions" stop criterion is likely to terminate the
search (can, of course, also be terminated if the optimal solution is detected and no cost increasing solutions are accepted). If, on the other hand, a continuous convergence (i.e., primarily makespan decreases, but also increases) towards the optimal solution is observed for a particular schedule, the "at most 400 proposed solutions" stop criterion is applied which in most cases will introduce a higher compile time overhead. Another stochastic phenomenon is the deadlock problem which gives rise to invalid solutions, and thus for some schedules more solutions have to be proposed in order to complete the Markov chains. Both of these circumstances affect the amount of CPU resources required to schedule a graph onto a given target architecture. Despite of these facts, table 6.6 actually provides some interesting results.

First of all, if we take a look at the time consumed by Hu*, NODUST, and DLS, we realize that these heuristics indeed are suited for inclusion in the Multiprocessor Scheduling Pool due to their very low compile time overhead. From the experimental results (i.e., for the actual set of DSP algorithms) it is obvious that Hu* and NODUST have almost (there are some deviations in tenth of seconds which are not reported in the table) identical run times, while DLS in all cases (the exception being the Cholesky factorization) has a longer run time. These observations can be explained by 1) the very similar node/processor selection procedures applied in Hu* and NODUST, and 2) the fact that for NODUST, a limited number of nodes (as compared to the total graph size) are actually duplicated (this can be seen from the Gantt charts). This leads to only a small additional overhead for NODUST. Secondly, the node/processor selection applied in DLS is exhaustive in the sense that all ready nodes are scheduled onto all processors (including IPC scheduling) in order to calculate dynamic levels for all node/processor combinations. This affects the compile time negatively. We observe in most cases a 2-3 times longer run time. However, as compared to the extreme run time increase discovered for the SA-based strategy, the DLS overhead is certainly neglectable.

Using an SA strategy, we observe (with no surprise) an extreme run time increase. We see, although, a somewhat similar pattern as found when scheduling the benchmark algorithms by the list heuristics.

It is evident that the graph size has an impact on the run time, but for the SA strategies we also realize that the graph topology plays an essential role. See e.g., the differences in run time required by the direct and the tree structured autocorrelation. The direct autocorrelation is characterized by a dominant sequential path which put some tight precedence constraints on the individual nodes leading to the earliest search termination of the two. The tree structured graph, although very regular, requires a longer run time. This may be explained by the high degree of parallelism which ask for much more combinations to be investigated before the termination can be activated.

Furthermore, if we compare the required run times for the Durbin and the Cholesky algorithms (which have approximately the same graph sizes and degree of parallelism), we see that the irregular Durbin algorithm requires at least nearly twice as long a run time as compared to the more regular oriented Cholesky algorithm. Again, we explain this difference by the higher slack (or flexibility) of many nodes in the Durbin algorithm which thereby calls for a higher run time overhead before the stop criterion can be meet.

From table 6.6 we clearly see the savings in run time which may be obtained if we select 1) SASGUM instead of SAS, and 2) LISA rather than SAS. The percentage savings are reported in table 6.7. As expected, LISA provides the highest run time reduction in all cases (except for the Durbin algorithm), but it is also worth noting the substantial improvements which may be obtained by SASGUM in most cases. It is, however, very difficult to extract some general conclusions from the SASGUM results, although it seems as graphs characterized by an irregular interconnection topology is gaining the highest saving, e.g., the Durbin algorithm and the wave
<table>
<thead>
<tr>
<th>Strategy</th>
<th>The Percentage Run Time Savings as Compared to SAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ROUX</td>
</tr>
<tr>
<td>SASGUM</td>
<td>10%</td>
</tr>
<tr>
<td>LISA</td>
<td>65%</td>
</tr>
</tbody>
</table>

Table 6.7: This table shows the percentage run time savings as compared to SAS for the set of benchmark DSP algorithms.

digital filter. This is also a reasonable indication if we assume that SAS works fairly well on regular graphs (the exception is the very regular recursive autocorrelation which actually shows the highest saving).

Similarly, it is almost impossible to set up any guide lines based on the LISA result. However, if we compare the percentage savings to the speedups in table 6.4, we recognize that those graphs for which a very high run time saving ($> 50\%$) is obtained (i.e., the Roux-Gueguen algorithm, the Cholesky algorithm, and the wave digital filter), are also graphs for which the initial DLS solution seems to be close to optimal (if we compared to the best results obtained by SAS/SASGUM). For these three cases, however, we also note that SAS has found equivalent or better solutions which therefore may indicate a too early invocation of the LISA stop criterion. If we, on the other hand, take a look at those cases where LISA has generated comparable or better solutions as compared to SAS/SASGUM (i.e. Cholesky, direct autocorrelation, recursive autocorrelation, and the FFT), we observe savings as high as $58\%$ and not below $15\%$. From this we conclude that LISA is a reliable SA technique which seems as an efficient choice when several SA based schedules have to be generated in order to alleviate the spreading problem.
Chapter 7

Conclusion and Perspective

We have now approached the final chapter of this thesis which means that it is time to draw the conclusions. We do this basically by reviewing our work and extract the results that we have obtained. On top of this, we propose a set of guide lines for selecting/using the strategies residing in our Multiprocessor Scheduling Pool. Finally, we put our work into perspective and suggest some ideas for future experiments and investigations.

7.1 A Review of Our Work

In chapter 1 we discussed the need for a design trajectory optimized for the experimentation with multi-PDSP solutions. Basically, we found that the overall design procedure includes numerous tasks among which we defined their interrelation.

Due to an exhaustive search through the literature, we concluded that very little progress actually has been made toward formulating a formal transformation of DSP (and in this work it means data independent) algorithms to a parallel program optimized for execution on multiple interconnected (primarily through a set of dedicated communication links) PDSPs. One exception, though, was the work by Lee & Messerschmitt, [29], who proposed the Synchronous Data Flow notations from which a Directed Acyclic Precedence Graph can be nicely constructed. However, we argued that their scheme do not produce realistic and efficient multiprocessor solutions, basically due to the fact that

- InterProcessor Communication is ignored in the scheduling phase, and

- only one simple scheduling technique is considered

From this important observation, we defined our domain of interest to be multiprocessor scheduling of data independent algorithms, and we defined two primary goals for our work;

1. Creation of a Multiprocessor Scheduling Pool including several realistic (in terms of IPC handling) strategies which may generate solutions of different efficiency for different types of precedence graphs (i.e., DSP algorithms).

2. From experimentations with these strategies applied to various graphs and architectures to formulate a set of guide lines for an efficient strategy selection.

A basic need for achieving the first goal was to formulate a set of realistic computational and architectural models. We found that researches previously for multi-computer types of architectures have proposed some models, [35], which, however, for our purpose are unable to handle
appropriately the IPC details. As a consequence, we reformulated in chapter 2 the computational
PWC (Precedence With Cost) model, and we devised a new formal scheduling notation to be
employed in the design of our scheduling strategies. Furthermore, for the actual implementa-
tion of the strategies we proposed general IPC based architectural models for shared memory and
message passing systems.

Our initial approach towards the design of strategies for the Scheduling Pool was based on
the simple Hu list scheduling heuristic as also suggested by Lee & Messerschmitt for their design
trajectory. We found that the plain Hu heuristic has some severe limitations, basically in terms
of a total absence of IPC handling. In chapter 3 we therefore proposed a set of modifications in
order to include

- realistic IPC handling, and
- efficient node- and processors selection.

We entitle this new list scheduling heuristic Hu*. Next, from [37] we got the idea that node
duplication in the presence of IPC may be a means to generate more efficient solutions (in terms
of shorter makespans). We therefore devised our NODUST list scheduling heuristic, which

- for all selected nodes detects whether the predecessors are potential for duplication, and
  next
- evaluates the set of predecessors potential for duplication in order to select the predeces-
  sor/processor pair which, in case of duplication, provides the earliest start time for the
  selected node.

Basically, NODUST is also a list scheduling heuristic where the individual nodes are se-
lected from a priority list. We created two versions of NODUST, 1) NODUST/ASAP, and
2) NODUST/CP. In the former, the priority list is generated by the means of an ASAP schedul-
ing, while in the latter the list is a Critical Path list (identical to the one employed in Hu*).

After we have reported on Hu* and NODUST in [57], Sih & Lee published their “Dynamic
Level Scheduler” (DLS) — a list scheduling heuristic where the priorities for node selection is a
function of

- the location of the node in the critical path
- the earliest time a given processor is ready to invoke the node
- the time at which all data tokens required by the node are available on a given processor
taking into account the status of the communication hardware

In [43], the DLS approach was reported to be the best known list scheduling heuristic of today
(we have not been able to trace more recent comparative studies which contradict with this
result), and we therefore decided to include it without any modifications in our Multiprocessor
Scheduling Pool.

From an exhaustive evaluation (in terms of speedup measures) of the four list scheduling
heuristics (Hu*, NODUST/ASAP, NODUST/CP, and DLS), based on randomly generated prece-
dence graphs with known communication/computation statistics, we can conclude the following.

- NODUST/CP is in most cases superior to 1) Hu*, and 2) NODUST/ASAP. For that reason,
  we decided to exclude NODUST/ASAP from further investigations.
• For a low IPC overhead, the improvement of NODUST/CP over Hu* is almost neglectable (typically less than 1-2%), whereas the improvements become distinct as the IPC is increased.

• DLS is in most case (i.e., for low, modest and high IPC) generating better solutions as compared to NODUST/CP. For a low IPC overhead, however, the improvement is typically less than a few percent.

• In the presence of ultra high IPC, NODUST/CP is always generating better solutions as compared to DLS in those interesting cases where the NODUST/CP speedup is higher than 1.

• For NODUST/CP and DLS, the speedup improvements observed when scheduling graphs with a high degree of parallelism as compared to graphs with a low degree of parallelism seems to level off by a factor of 2 then the graph size is doubled.

The overall conclusion which can be drawn from the experimentation with the list scheduling heuristics applied to random graphs is therefore that Hu*, NODUST/CP, as well as DLS are all worthy members of the Multiprocessor Scheduling Pool, since they for different 1) IPC overheads, 2) graph sizes and -topologies, and 3) number of processors may individually provide the best solution.

Now, caused by our recognition of the good DLS performance (which is consistent with the results reported in [43]), we started a search for other new methods which in general are able to outperform the DLS technique. Our hunch is that the development of new list scheduling heuristics may lead to further improvements, e.g., by incorporating the NODUST concept into the DLS philosophy, but still we conclude that it may be difficult to obtain significantly better results. The reason is that DLS in fact is performing an exhaustive evaluation of all ready nodes on all processors. We therefore took a somewhat different approach.

The list scheduling heuristics are suffering from the horizon effect which means that they are searching only partially through the solution space. Since our scheduling problem is basically a combinatorial optimization problem, we turned in chapter 4 our attention into Simulated Annealing, a method which previously has been successfully employed to solve related problems, such as The Traveling Salesman Problem, place & route of VLSI circuits, etc.

We also found that D'Hollander, [49], previously has used simulated annealing for scheduling of directed task graphs in multi-computer environments. We argued that his work, however, has some severe limitations in its ability to search the total solution space (due to a critical path based node selection procedure) and thus the optimal solution may easily be excluded. Furthermore, from a number of other disadvantages characterizing D'Hollander's method we concluded that the development of a new strategy optimized for our purpose was highly relevant, and thus we devised the Simulated Annealing Scheduler, SAS.

Comparing SAS against NODUST/CP and DLS in terms of speedups derived from scheduling the set of randomly generated precedence graphs, we can conclude the following.

• SAS is in most cases superior to NODUST/CP and DLS. SAS becomes more outstanding the higher the IPC overhead.

• SAS seems to perform relatively better on graphs with a high degree of inherent parallelism.

• A careful tuning of the SAS parameters, basically the Markov chain length, the distance parameter, and the stop criterion is crucial in relation to the graph size and the number of processors.
In the case of heavy IPC overhead, SAS is 1) in most cases able to do better than NODUST/CP, and 2) always, except for sequential types of graphs, generating parallel solutions with a speedup larger than 1.

These are all very nice properties but we also found (due to its stochastic nature) that SAS may get trapped in a local optimum. As a consequence, the generation of multiple schedules of the same precedence graph using identical sets of parameters may lead to different solutions. We can conclude, that this problem may be solved by a sufficient 1) decrease of the distance parameter, and 2) increase of the Markov chain length. The penalty, however, is a substantial run time increase which is certainly unwanted.

In chapter 5, we therefore proposed a number of optimizations to the SAS strategy. As a first approach, we suggested an extended node move function which takes into account the IPC and thereby guides the move node to the processor with which it has the highest input communication overhead. We entitled this approach “SAS with GUid ed Move”, SAGUM, and from numerous experiments with random graphs it can be concluded that

- SASGUM seems able to generate better solutions (in terms of speedup) as compared to SAS (for identical sets of SA parameters) for mostly sequential type of graphs. This holds for 1) low as well as for high IPC, 2) any number of processors, and 3) several selected degrees of guided move. The random graph experiments gave no clear indications of the performance in case of a high degree of inherent parallelism.

- The spreading on the SASGUM results is not significantly different from the SAS spreading. This means that SASGUM in some cases is also trapped in a local optimum.

Due to the latter point, we suggested SASGUM.IO which guides the move node according to 1) input IPC from predecessors, as well as 2) output IPC to successors. In this work, unfortunately, we have not been able to conduct any experiments in order to verify this approach.

Finally, we recognized that a major part of the solutions proposed as required to “travel” from the random initial to the final (optimal) solution is actually spend in getting close to a near-optimal solution. We therefore suggested the LISA strategy, a hybrid LIst SA scheduler, which launches its SA search from the list generated DLS solution. Preliminary results showed that LISA 1) may further optimize the near-optimal DLS solution, and 2) reduce significantly the required run time.

### 7.2 A Set of Scheduling Guide Lines

In order to 1) further broaden the scope of our work, and 2) pave the way for a reliable formulation of strategy selection guide lines, we created a set of DSP benchmark algorithms characterized by different 1) graph sizes, 2) degrees of inherent parallelism, and 3) graph topologies.

Chapter 6 thoroughly elaborates on the algorithms selected for the benchmark library, and it describes our scheduling experiments and results based on models of fully connected 2-, 4-, and 6-processor ADSP-2106x message passing architectures. Beside, we measured the actual run time required by the various strategies to terminate. Furthermore, we extract essential characteristics from the scheduling results, and these observations are now used in order to formulate a set of guide lines for the selection and application of the strategies residing in our Multiprocessor Scheduling Pool.
CHAPTER 7. CONCLUSION AND PERSPECTIVE

For the system designer, the first question which naturally needs to be answered is; "Given 1) an algorithm (i.e., a DAPG) with i) a certain degree of parallelism/grain size, and iii) interconnection topology, and 2) an architecture with a specific i) number of processors, and ii) communication facility, is it realistic at all to go for a parallel solution?" In order to give a fast and reliable answer, we suggest to select one of the list scheduling heuristics. Specifically, the actual choice may be subject to the following rule:

- For algorithms with 1) a very low degree of parallelism, or 2) a very limited IPC overhead, the Hu' strategy seems as a reasonable "first choice".

- If, on the other hand, 1) the degree of parallelism is high, 2) the amount of potential IPC overhead is comparable to or higher than the amount of computations, and 3) the interconnection topology is non-trivial, a better choice may be NODUST or DLS, primary depending on the IPC overhead.

Based on the result (in terms of speedup, processor utilization, storage requirement, etc.) provided by this initial experiment, there are several options. One could suggest to change 1) the grain size by the use of a clustering/declustering algorithm, or 2) the number of processors in the target architecture, and then redo the list scheduling, possibly in terms of another strategy as compared to the first trial.

Alternatively, if it seems appropriate to create a parallel solution under the initial constraints, one may next choose to further optimize the list scheduling solution by experimenting with the SA based strategies. Again, in order to investigate what may actually be possible to gain, we suggest a low complexity SA strategy, i.e., LISA.

Based on the actual 1) graph size, 2) interconnection topology (regular/irregular), and 3) number of processors, a set of SA parameters has to be selected. We conclude that more experiments are needed before we suggest any definite guide lines, but roughly speaking it seems as for 1) a small graph size (i.e., less than 20-40 nodes), 2) a small number of processors (i.e., a most 2-3), and 3) graphs characterized by a limited node flexibility (i.e., graphs where most nodes have only a small slack),

- the Markov chain length can be safely chosen equal to \( N \). Exceeding these limits, \( 2N \) or more is recommended.

- A distance parameter equal to 0.25-0.50 seems to suit most cases, although we in general suggest to lower the value the more complex the scheduling problem is.

- It seems as though the IAR value should be selected "higher" the more complex the scheduling problem in order to escape efficiently from the DLS solution. For more simple problems, 1-2\% has proven sufficient.

Since LISA (in the current implementation) is SAS based, we cannot guarantee that the "optimal" solution is found in the first trial (although our investigations are showing that LISA is doing a good job). A reasonable strategy is therefore to redo the LISA experiment. Assuming that LISA in one of these trials provides a "significant" improvement over the DLS solution, we have reasons to believe that a more exhaustive SA search in terms of SAS/SASGUM may generate an even better solution.

The knowledge we have obtained so far indicates that SASGUM is a better strategy as compared to the plain SAS, since in most cases (here we refer to the benchmark examples) it 1) finds comparable or better solutions, and 2) terminates earlier (this conclusion is based on a \( \gamma \)-value
equal to 0.5, but so far we have not been able to formulate any guidelines as to how to select appropriately this parameter. We therefore suggest that SASGUM is employ in a first trial, especially if 1) the graph has limited parallelism, or the number of processors is low.

It is evident that if one chooses to employ SAS/SASGUM, and in particular for complex problems, the run time may be rather long. It means that we are interested in doing the scheduling only once, and thus it seems as a good strategy to select 1) the Markov chain length, and 2) the distance parameter longer, respectively smaller than actually suggested above, of course, in order to avoid quenching as thereby get stucked in a local optimum.

Alternatively, the SA parameter selection can be relaxed somewhat thus lowering the run time providing an argument for several SAS/SASGUM trials.

We emphasize that these guidelines are based only the experiments reported in this thesis, and we therefore expect some modifications to be inevitable as our knowledge is enlarged when additional experiments with 1) other architectural types, 2) communication schemes, 3) DSP algorithms, and 4) SA parameters have been performed.

As an overall conclusion we state that our Multiprocessor Scheduling Pool and the corresponding set of guidelines is an outstanding tool which can be very efficiently employed by DSP system designers in order to carry out numerous experiments, and thereby gain valuable knowledge before the actual multiprocessor assembly program is created.

7.3 A Perspective View

In order to close this thesis, we finally propose some ideas for extensions and future work which may be interesting in terms of further improvements to 1) the overall design trajectory as such, and 2) the individual scheduling strategies.

An urgent need for the scheme to be more user friendly is the inclusion of a parser which can transform a textual executable specification into the SDF notation. As long as this tool is absent, the designer has to hand craft the data flow graph himself which, of course, is practically achievable (as we have demonstrated by numerous examples) but rather time consuming and certainly not optimal for larger applications.

We argued in chapter 1 that a proper grain size (and thus a suitable trade-off between the computational and the IPC overhead) is essential for obtaining the optimal solution, and through our case studies we have illustrated this by varying the communicational amount in the individual graphs/architectures. We have basically assumed that a grain size adjust mechanism is available for application in our scheme if necessary, but so far we have not paid any special attention to the subject. We note, however, that researchers have done some work in this area. Examples are [58, 59]. An obvious extension to the current version of our design trajectory is therefore to adapt one or more of these existing clustering methods to interface to our DAPG format. The inclusion of a clustering/declustering mechanism will provide an exceptional possibility to perform more exhaustive experimentations with the individual DSP algorithms, and thus further supporting our primary goal of realistic and efficient multi-DSP design.

As we have briefly mentioned during the investigations of the scheduling strategies, additional improvements may be possible. We saw, for the example, that the SASGUM philosophy seems appropriate in many cases, and we have suggested a further extension in terms of the SASGUM IO strategy. Obviously, before we can draw any conclusions to whether we can benefit in some situations from this IO-based approach, a series of experiments have to be conducted.

Furthermore, our LISA technique has been implemented only by the use of the plain SAS algorithm. One could suggest that improvements in terms of a further run time reduction may be possible by merging SASGUM into the LISA approach.
Although we have conducted a substantial amount of experiments with different graph topologies and sizes, IPC overheads, and number of processors in the target architecture, we suggest that even more examples are investigated in order to provide more guidelines for selection of the SA parameters.

Finally, we would like to mention that an alternative method for suitable solving of the combinatorial optimization problem has become popular during the time when this thesis has been in preparation. People have now started to look into the application of Genetic Algorithms (GA), [60], which basically tries to generate the optimal solution from a "survival of the fittest" principle. It is out of scope here to discuss in detail the working of GA and its application to scheduling, but we would like to point out that we have conducted some preliminary experiments, [61], which show that GA may be a worthy supplement to our SA strategies. GA is also an iterative method, and thus it basically inherits the same problems as found in SA in terms of 1) extensive run time overheads, and 2) the spreading problem. A hot topic for future research is therefore to investigate exhaustively the performances of GA scheduling, compare it to our SA results, and devise new GA based techniques, e.g., a LIGA approach, in order to provide even more opportunities for the DSP system designer to select "the ultimate" scheduling strategy for his specific problem.
Bibliography


BIBLIOGRAPHY


Appendix A

Preprocessing the DSP Algorithms

The problem of mapping an algorithm onto a multi-computer architecture has during recent years been addressed by numerous researchers, e.g., [62, 63, 64]. Common for most of these works is that the algorithm initially is converted into a graphic representation — a problem graph — where the graph nodes are the algorithmic subtasks and the graph arcs represent the precedence relations among the nodes. Similarly, the target architecture is also described in terms of a graph — a system graph — where each node corresponds to an autonomous processing element (PE). In the system graph, the edges are physical communication links between the PEs, and thus the system graph is a 1:1 description of the actual multi-computer architecture. The aim is next to map the problem graph onto the system graph such that a specific objective function is optimized.

In this work, we focus on target architectures with multiple PDSPs interconnected via 1) dedicated communication links (as provided by the most recent processors), or 2) a shared memory. We consider the PDSPs as processing elements which are able to execute any operation (i.e., node) in the problem graph. The operations and their interrelation must be specified in the problem graph which we will derive using techniques based on previously proposed ideas. Such a graph is particularly essential because all information of the algorithm's inherent parallelism is naturally exploited.

In order to prepare for scheduling of DSP kind of algorithms onto multi-PDSP architectures, we in this appendix discuss 1) the DSP-optimized Synchronous Data Flow Graph notation proposed by Lee & Messerschmitt [29], and 2) our algorithms for appropriate transformation of an SDFG into an DAPG.

A.1 Signal Flow and Data Flow Graphs

The graphical representation of a DSP algorithm has two primary purposes.

- It is much easier to explore the inherent parallelism in an algorithm represented as a graph as compared to a textural description.

- A graph is a natural input format for scheduling

We are therefore need an appropriate graph notation which may be applied as input for the scheduling strategies.
A.1.1 The Signal Flow Graph Notation

In the DSP community, a broad class of algorithms are expressed as a signal flow graph (SFG) [65, 53] since it provides a natural and convenient medium for design, modification and representation. Formally, an SFG is a network of directed arcs that connect at nodes. Associated with each node is a variable or node value. An example is shown in figure A.1.

![Signal Flow Graph](image)

Figure A.1: An example of a signal flow graph with two nodes and five arcs. For simplicity, only one arc is fully specified.

Signals arriving at node $j$ are added together and next represented by the node value $w_j[n]$. A node may receive multiple input signals and it may provide the node value on multiple output arcs. The signal propagating between node $j$ and node $k$ is “modified” by a time function $f_{jk}[n]$, which may be e.g., a multiplication. The important thing here is, however, that the SFG notation is characterized by activities on arcs. This is in contrast to the general data flow graph (DFG) notation where the activities are on the nodes [66].

A.1.2 The Data Flow Graph Notation

An DFG is a directed graph (the token on each arc flows in one and only one direction [67]) whose nodes correspond to operations, and arcs are pointers for forwarding tokens. The graph demonstrates sequencing constraints (i.e., precedences) among the nodes — a node cannot start execution before all tokens on its input arcs are available which in any practical situation means that all its predecessor nodes (henceforth predecessors) have run to completion. A simple example DFG of the calculation $z = \sqrt{xy} + (xy)^2$ is shown in figure A.2. Here, the addition cannot start until both the square and the square root have run to completion. Similarly, these operations depend on their predecessor – the multiplication.

![Data Flow Graph](image)

Figure A.2: The data flow graph notation is characterized by activities on nodes — here illustrated by a simple mathematical example.

In general, however, the DFG nodes may involve data dependent calculations, i.e. constructions where a token supplied from an input arc controls the behaviour of the node. In such cases, the actual number of tokens on some arcs may be unknown before execution. Since data
dependent calculations are rare (but not totally absent) in DSP- and other scientific algorithms, we have opted for the SDF notation which efficiently expresses graphs with exactly known production and consumption of tokens.

A.2 The Synchronous Data Flow Notation

The SDF notation differs from traditional data flow in that the amount of tokens produced and consumed by each node is specified a priori for each in- and output of all nodes — therefore the name synchronous. For DSP applications this is equivalent to specification of the relative sample rates for the individual subalgorithms which are represented by the graph nodes. The primary advantage of this a priori knowledge is that static scheduling of the graph may be possible.

SDF can handle a broad range of grain sizes although initially developed as a technique for describing and programming large grain data flow graphs. Similar to traditional data flow notation, the activities are on the nodes. A SDFG may therefore be regarded as a block diagram description of the DSP application algorithm — a natural and flexible environment for design and expression of a very broad class of algorithms. The blocks (nodes) themselves can be described by a SDFG with a lower granularity (grain size), and thus a hierarchy can be easily constructed.

![Figure A.3: In the synchronous data flow notation, the number of tokens produced and consumed on every arc are known a priori.](image)

An example of an SDFG is shown in figure A.3. Each node is given an identification number and to each arc is associated numbers representing the amount of tokens produced and consumed, respectively, on that particular arc. As we shall soon recognize, the number of tokens produced onto a specific arc may not necessarily equal the number of tokens consumed from that arc. Related to the example SDFG on figure A.3, e.g., c and g are in general not equal. As a special case, the graph is said to be homogeneous if all nodes produce and consume exactly one token on all arcs. This situation frequently occurs in many DSP algorithms, e.g., in digital filters. In case the numbers of tokens produced and consumed are not identical, the solution is to execute (or evoke) the involved nodes an unequal number of times. This, of course, has an impact on other nodes in the graph directly or indirectly connected to the involved nodes. Theories for efficient solution of this problem have been developed, and we will shortly refer to them. Also buffering of tokens between nodes is essential in order to collect sufficient data for a node to be evoked. Theories for practical handling of the buffering problem will not be discussed here, but can be found in [68].

Our overall purpose is to discuss conversion of the SDFG into an DAPG. The first step towards the precedence graph is to construct a Periodic Admissible Sequential Schedule (PASS).
The schedule has to be periodic since we are focusing on DSP algorithms which in theory are repeated forever. "Admissible" means that the nodes will be scheduled only when data are available, and that a finite amount of memory (to hold tokens between nodes) is required. The *sequential schedule* is simply a special case of the parallel schedule — only one processor is required. Therefore, once a sequential schedule is found, we know that a parallel version may exists. On the other hand, if an PASS cannot be found, it doesn't make sense to spend any effort in searching for a parallel solution.

It should be noted that there are some limitations with the SDF notation. One limitation concerns nodes having only inputs which are constrained by being supplied from the outside world, e.g., from an AD-converter. It is impossible to specify exactly when such nodes have sufficient input data available and when they are ready to be evoked. An assumption is therefore that such nodes always have sufficient data available, and hence their input arcs are simply removed. In practice, however, this cannot be true although for scheduling purposes this assumption seems to apply without problems. To our knowledge, nothing have been reported in the literature on this limitation. Concerning nodes producing tokens to the outside world, e.g., to a DA-converter, we similarly assume that data are supplied whenever produced and thus the output arcs on these nodes can be ignored. Using the example SDFG from figure A.3, this graph is therefore simplified as illustrated in figure A.4.

![Graph](image)

Figure A.4: Assuming that input from and output to the external world are always available, the SDFG can be simplified by removing the 1/O-arcs.

In order to derive a formal method for SDFG-to-DAPG conversion, the SDFG must next be represented in matrix notation — similar to traditional directed graphs [67]. The matrix employed in this case is known as the *Topology Matrix*, and is denoted by $\Gamma$. The topology matrix can be constructed only if all arcs are also given an identification number as shown in figure A.5.

The topology matrix is then made by assigning a column to each node and a row to each arc. By this notation, the $(i, j)^{th}$ entry in the matrix then denotes the amount of tokens produced by the $j^{th}$ node on the $i^{th}$ arc each time node $j$ executes. If node $j$ consumes tokens from arc $i$, the entry is simply negative, and zero if node $j$ is not connected to arc $i$.

For our example SDFG, the topology matrix, which in general do not need to be square, is defined as

$$
\Gamma = \begin{bmatrix}
  b & -d & 0 \\
  c & 0 & -g \\
  0 & -e & i \\
\end{bmatrix}
$$

(A.1)

The topology matrix is next extensively employed in order to determine
Figure A.5: In order to construct a topology matrix, all arcs are identified by a specific number.

- the amount of tokens on each individual arc during execution of the graph
- how many times each node should be invoked during one sample period
- the sequence of node invocations

The actual number of tokens on every arc at any given point in time, \( n \), can be described by a buffer vector, \( b(n) \), which has one element for each arc. For the purpose of deriving a sequential schedule, it doesn’t matter how long time the individual nodes actually execute. The only thing which is of importance here, is the ordering among nodes. The “time index”, \( n \), may therefore be updated whenever a node finishes execution.

Similarly, the node which actually starts execution a time \( n \) may be identified by an invocation vector, \( v(n) \), which has one element for each node. All elements in \( v(n) \) equal zero except the element corresponding to the node which is currently executing — this element equals 1.

With these definitions, it is possible to express the changes in the amount of tokens on the arcs imposed by invocation of a node.

\[
b(n + 1) = b(n) + \Gamma v(n) \tag{A.2}\]

In this equation, we assume an initial amount of tokens on the arcs represented by \( b(0) \). For some DSP algorithms, initial values are present on the arcs. In digital filters, for instance, the unit sample delays actually correspond to initial tokens on some arcs — typically reset to zero before execution of the first sample period. The SDF notation is therefore capable of handling delays. They are expressed in the SDFG by “\( x \cdot D \)” on arcs holding a total of \( x \) initial tokens. In a SDFG with a single delay between node \( A \) and node \( B \), the token produced by \( A \) at time \( n \) will be consumed by \( B \) at time \( n + 1 \). In SDFGs which are cyclic (include loops) and which receive no data from the outside world, there is a specific requirement for delays (initial tokens) in order to prevent the dead lock situation — in other words, there must be sufficient initial data such that at least one of the nodes can start execution.

It can be shown (which is not trivial) that a necessary condition for the existence of a PASS is that

\[
\text{rank}(\Gamma) = s - 1 \tag{A.3}\]

where \( s \) is the number of nodes in the SDFG (\( s \) should not be confused with our usual notation of a parallel schedule). Some of the partial results obtained when deriving this equation are very illustrative and we therefore elaborate on them here.
Assume that an PASS can be represented by an ordered list, $\phi$, of all node invocations required in order to execute one sample period of the SDFG. For now we do not bother about how the actual number of invocations for the individual nodes have been obtained. We denote the number of items in $\phi$ by $p$ and define this number as the period. From equation A.2 it is evident that we can write

\[ b(1) = b(0) + \Gamma v(0) \]  
\[ b(2) = (b(0) + \Gamma v(0)) + \Gamma v(1) \]  
\[ b(3) = (b(0) + \Gamma v(0) + \Gamma v(1)) + \Gamma v(2) \]

and so on, until all $p$ invocations have been performed. If an instance vector $q$ is defined as

\[ q = \sum_{n=0}^{p-1} v(n) \]  
then we may write

\[ b(p) = b(0) + \Gamma q \]  

Here we note that $q$ represents the actual number of invocations for all nodes. By definition, a PASS is periodic (exactly the same schedule is repeated infinitely), and thus we have

\[ b(np) = b(0) + n\Gamma q \]

since at the $n^{th}$ period, the individual nodes have been evoked exactly $nq$ times.

Previously, we stated that for a PASS, a limited amount of memory is required in order to execute it infinitely many times (admissible). From equation A.9 it is thus evident that the memory holding the tokens between the nodes can be bounded if and only if

\[ \Gamma q = 0 \]

where 0 is the zero-vector.

Of course, this equation has the trivial solution which, however, characterize the situation where no nodes are ever invoked. Therefore, $q \neq 0$. Since $q1$ is in the nullspace of the topology matrix, and 2) has the dimension $s$, it can be shown that

\[ \text{rank}(\Gamma) < s \]

and therefore the rank of $\Gamma$ equals $s - 1$ for an PASS.

Based on this discussion we see that it is necessary to solve the characteristic equation (equation A.10), and the solution ($q$) specifies how many times (per sample period) each node in the SDFG has to be executed so that no tokens will heap up on the arcs. Note that $q$ is not an unique solution to equation A.10 — all vectors $Jq$, where $J$ is an integer, are also a solution (we assume that $q$ holds only integer elements). In order to specify the number of invocations of each individual node in one period, the solution is the invocation vector which has the minimal integer elements. Henceforth we denote this solution $q_{min}$.

It should also be evident that after each complete period, exactly the same number of tokens reside on the arcs as initially specified by $b(0)$. 

APPENDIX A. PREPROCESSING THE DSP ALGORITHMS

138
A.3 An Algorithm for PASS Generation

According to the theories derived in [29], there exists a class of algorithms, which, given a positive vector \( q \) satisfying equation A.10, find a PASS with period \( p \) equal to the sum of the elements in \( q \) — and in this work we mean \( q_{\text{min}} \).

Since we have opted for the SDF notation as input for our scheme we need to design such an algorithm which on the basis of

- the topology matrix \( \Gamma \), and
- the initial buffer vector \( b(0) \)

generates a PASS, if it exists.

In order to determine when the individual nodes are able to be invoked, a classification into external and internal nodes is required for our algorithm. We therefore extend the SDFG notation by defining 1) an external node as a node which has only inputs from the outside world (and thus according to the above discussion have no input arcs), and 2) an internal node as a node which has one or more inputs supplied from other nodes in the graph (in chapter 6, we make a further subdivision of internal nodes). From the topology matrix definition and the SDFG example in equation A.1, it is clear that only nodes whose corresponding topology matrix column holding entirely zeros and/or positive numbers can be external nodes — all other nodes are classified as internal.

Before describing our proposal for an PASS-algorithm, we need to define a few variables; again \( s \) denotes the number of nodes in the SDFG (corresponding to the number of columns in \( \Gamma \)), and \( t \) represents the number of arcs (corresponding to the number of rows in \( \Gamma \)). Each node in the SDFG is given an identification number which is referred to as \( N_{\alpha} \). The devised PASS-algorithm is described in pseudo code in algorithm 17.

Initially, the rank of \( \Gamma \) is calculated in order to check the necessary condition for the existence of a PASS. Next, all columns of \( \Gamma \) are evaluated in order to classify the nodes as either inner or external, and the minimal number of invocations of each node is determined by solving the homogeneous \( \Gamma q \)-equation. This provides the necessary information for calculating the period \( p \).

Now, all nodes in the SDFG are represented in a list \( L \). The ordering of the nodes in this list has a substantial impact on the generated PASS in case the algorithm represented by the SDFG is not strictly sequential. However, since the overall purpose is to derive a precedence graph, the PASS can be seen as an intermediate result which do not influence the final DAPG, and hence \( L \) can be constructed arbitrarily. We assume that the nodes in the SDFG are assigned identification numbers randomly which means that a numerically ordered \( L \) (as shown in algorithm 17) represent the desired arbitrary list. It can be shown that a random ordering of the columns in \( \Gamma \) (i.e. a random assignment of SDFG node numbers) do not have any impact on the equations for determination of PASS-existence.

Now, the purpose of the while-loop is to

1. detect runnable nodes (i.e., nodes having sufficient input tokens) from \( L \), and
2. assign these nodes to the list \( \phi \).

Basically, the algorithm run through all nodes in \( L \). If the selected \( N_{\alpha} \) is an internal node, then it is only runnable if all its predecessors have executed the number of times sufficient to
Calculate \( \text{rank}(\Gamma) \);
If \( \text{rank}(\Gamma) \neq s - 1 \) then no PASS exists \( \Rightarrow \) terminate;
On the basis of \( \Gamma \) determine which nodes are internal and external, respectively;
Find the non-trivial solution to \( \Gamma q_{\text{min}} = 0 \);
Determine the period as \( p = n \sum_{i=1}^{s} q_{i,\text{min}} \);
Create the list \( L = \{N_1, N_2, N_3, \ldots, N_s\} \) of all the nodes in the SDFG;
Initialization: Counter \( i = 0 \), the instance vector \( q(0) = 0 \), and the list of sequential evocations \( \phi = \emptyset \);
While \( i \leq p - 1 \) do
  \{ 
  Initialize \( s \) lists defined for input arcs to node \( N_\alpha \): \( L_{\alpha,\text{input}} = \emptyset \);
  For \( \alpha = 1 \) to \( s \) 
    \{ 
    Select \( N_\alpha \) from the list \( L_\alpha \);
    If \( N_\alpha \) is an internal node then
      For \( l = 1 \) to \( t \)
        If \( \Gamma_{\alpha l} < 0 \) then add \( l \) to the list \( L_{\alpha,\text{input}} \);
        If \( ((l \in L_{\alpha,\text{input}}) \land (b_l(i) \geq -\Gamma_{\alpha l}) \lor (N_\alpha \text{ is an external node})) \land (q_{\alpha l}(i) < q_{\alpha,\text{min}})) \) then 
          \{ 
          Add \( N_\alpha \) to the list \( \phi \) of sequential node evocations;
          Update the invocation vector: 1) \( v(i) = 0 \), and 2) \( v_{\alpha l}(i) = 1 \);
          Update the buffer vector: \( b(i + 1) = b(i) + \Gamma v(i) \);
          Update the instance vector: \( q(i + 1) = q(i) + v(i) \);
          Increment invocation counter: \( i + 1 \);
          \}
    \}
  \}
If \( b(i) = b(0) \) then PASS=\( \phi \) else terminate with error;
end.

Algorithm 17: The proposed algorithm for PASS generation.

produce the required amount of input tokens. Therefore, we first detect all the input arcs to \( N_\alpha \) and place these arcs in the list \( L_{\alpha,\text{input}} \).

\( N_\alpha \) is runnable if 1) the amount of tokens on the input arcs represented by \( b \) is greater than or equal to the amount of data it consumes (specified by the appropriate entry in \( \Gamma \)), or if 2) \( N_\alpha \) is \( i \) external and \( ii \) has not executed the number of times specified by \( q_{\text{min}} \).

When \( N_\alpha \) has been detected as being runnable, it is assigned to \( \phi \) and all the relevant vectors, as well as an invocation counter \( i \), are updated. When this counter \( (i) \) equals \( p_\alpha \), all nodes in the SDFG have been assigned to \( \phi \).

Since \( \text{rank}(\Gamma) \) can only be applied as a necessary, but not sufficient check for the existence for a PASS, we finally verify that the amount of tokens on the arcs after one period is equal to the initial amount of tokens. If this is true, the PASS equals \( \phi \), otherwise there is an error in the SDFG and we simply terminate the procedure.

As pointed out previously, the existence of a PASS indicates that a parallel schedule may be found. Deriving such a schedule involves several steps where the initial one is to create a Directed Acyclic Precedence Graph (DAPG) — the input notation applied by the scheduling strategies. Deriving such a graph is discussed in the following section.

A.4 Constructing an DAPG

The DAPG construction serves two purposes; First of all, the SDFG may be cyclic which cannot be handled by the scheduling strategies. As a consequence, a cyclic SDFG has to be made
acyclic, which in fact is a side effect obtained through construction of the DAPG. Secondly, for the given grain size specified by the initial SDFG, the DAPG indicates the precedences among the individual graph nodes. Therefore, the scheduling strategy can on the basis of the inter-node precedences expressed by the DAPG select the next node (i.e., a node whose predecessors have already been scheduled) and schedule this node onto an idle processors.

Since the DAPG represents the DSP algorithm on a form applicable as input for the scheduling strategies, the number of sample periods over which the schedule is to be made, must be explicitly expressed in the DAPG. An improvement in makespan may be obtained by increasing the number of sample periods over which the schedule is made. Since no formal techniques for determination of the optimal number is known, heuristics are required. A common method is simply to increase the number of sample periods until no further improvement can be registered in the generated schedule. This procedure, however, requires that for each evaluation of the makespan, a DAPG generation and a complete schedule has to be made. For highly complex DSP algorithms, this procedure may therefore be rather time consuming and the improvement obtainable may not be comparable with the effort needed in order to determine the optimal number of sample intervals. In this work, we therefore devise our methods on the basis of a single sample interval, knowing that better solutions may be found by increasing the sample interval count.

The algorithm we have constructed for DAPG extraction has many similarities as compared to our algorithm for PASS generation. Again the initial purpose is to detect runnable nodes from an arbitrarily ordered list of all the nodes in the SDFG. A detected runnable node is added to the DAPG and precedence links to appropriate predecessors are next established. Our DAPG-algorithm is shown in pseudo code in algorithm 18.

Initially, the DAPG-algorithm is doing the same as the PASS-algorithm — a runnable node is detected. Next, it is added to the DAPG. Since the DAPG represents the precedences among the nodes, the algorithm has the ability to detect the required number of precedence links and assign these links from the current node to its predecessors.

The precedence link detection/assignment procedure works as follows; If the node detected as being runnable is an external node, then, of course, it doesn’t have any predecessors and no precedence links are therefore required. This means that the invocation vector, the buffer vector, the instance vector as well as the invocation counter simply can be updated and the next runnable node can then be detected. On the other hand, if the node (henceforth \( N_\alpha \)) is an internal node, then we have to find all predecessors to \( N_\alpha \). These predecessors will be named \( N_\beta \) in the following. Locating an \( N_\beta \) is done by searching through all \( t \) arcs in the SDFG — which is implemented as a search through all entries in column \( \alpha \) of \( \Gamma \). A negative entry (in row \( j \)) indicates that \( N_\alpha \) has a predecessor connected to arc \( j \). Since only one node produces tokens onto arc \( j \), then there must be one and only one positive entry in row \( j \). The column corresponding to this positive entry is column number \( \beta \) — and thus we have found the number of the node which produces data onto arc \( j \), and thereby the predecessor to \( N_\alpha \), i.e., \( N_\beta \).

Next, it is essential to determine the actual invocation \( k \) of \( N_\alpha \). Since entry \( \alpha \) in the instance vector \( \xi \) indicates how many time \( N_\alpha \) has already executed, \( k \) is obtained by an increase of this value. Now, in order to execute \( N_\alpha \) for the \( k^{th} \) time, the appropriate amount of tokens must be available. In other words, the \( k^{th} \) invocation of \( N_\alpha \) requires precedence links to a specific number of predecessors \( N_\beta \). This number is given as \( d \) in the algorithm. Note first of all, that the number of tokens required in order to execute \( N_\alpha \) \( k \) times equals \(-k \cdot \Gamma_j\alpha\). A total of \( b_j(0) \) of these tokens are provided initially on the arc \( j \). Therefore, if \( b_j(0) \geq -k \cdot \Gamma_j\alpha \), we conclude that the \( k^{th} \) run of \( N_\alpha \) do not depend on \( N_\beta \). However, if this is not the case, \( N_\beta \) must produce \(-k \cdot \Gamma_j\alpha - b_j(0) \) tokens before the \( k^{th} \) run of \( N_\alpha \). On every invocation of \( N_\beta \), a total of \( \Gamma_j\beta \)
Form an arbitrary list: $L = N_1, N_2, \cdots, N_s$;
Initialization: counter $i = 0$, and the instance vector $q(0) = 0$;
While $q(i) \neq q_{\text{min}}$ do
  
  Initialize $s$ lists defined for input arcs to node $N_\alpha$: $L_{\alpha, \text{input}} = \emptyset$;
  For $\alpha = 1$ to $s$
    
    Select $N_\alpha$ from the list $L$;
    If $N_\alpha$ is an internal node then
      
      For $l = 1$ to $t$
        
        If $\Gamma_{l\alpha} < 0$ then add $l$ to the list $L_{\alpha, \text{input}}$;
        
        If $[[\forall l \in L_{\alpha, \text{input}}, b_l(i) \geq -\Gamma_{l\alpha}] \lor (N_\alpha \text{ is an external node})] \land (q_\alpha(i) < q_{\alpha, \text{min}})]$ then
          
          Add an instance of $N_\alpha$ to the DAP-graph;
          
        If $N_\alpha$ is an internal node then
          
          For $j = 1$ to $t$
            
            If $\Gamma_{j\alpha} < 0$ then
              
              Find the positive entry in row $j$ of $\Gamma$;
              
              The corresponding column number $\beta$ equals the node number of the predecessor $N_\beta$ to $N_\alpha$ on arc $j$;
              
              Calculate the actual run $k$ of $N_\alpha$: $k = q_\alpha(i) + 1$;
              
              Calculate the actual number of precedence links $d$: $d = \lceil \frac{-k \Gamma_{j\alpha} - b_j(i)}{\Gamma_{j\beta}} \rceil$;
              
              If $d < 0$ then $d = 0$;
              
              Make precedence link(s) from $d$ instances of $N_\beta$ to $N_\alpha$;
            
          
          Update the invocation vector: 1) $v(i) = 0$, and 2) $v_\alpha(i) = 1$;
          
          Update the buffer vector: $b(i + 1) = b(i) + \Gamma v(i)$;
          
          Update the instance vector: $q(i + 1) = q(i) + v(i)$;
          
          Increment invocation counter: $i + +$;
        
    
  
End.

Algorithm 18: The proposed algorithm for DAPG generation.
tokens are produced onto arc \( j \). From this, it is evident that the \( k^{th} \) run of \( N_\alpha \) depends on the first \( d \) runs of \( N_\beta \), where \( d \) is calculated as shown in algorithm 18.

Based on this calculation it may turn out that \( d \) is lesser than zero. In such cases, \( d \) is forced to zero and thus no precedence links are needed. Otherwise, \( d \) precedence links are established from \( N_\alpha \) to \( d \) instances of \( N_\beta \) already present in the DAPG.

Finally, the usual vector updates are performed. The procedure is simply repeated until all nodes have been represented in the DAPG the number of times they actually have to execute — this is implemented by testing on the minimum integer vector in the zero-space of \( \Gamma \), \( q_{\min} \). After termination, the DAPG is completed — 1) all nodes are represented in the graph, 2) the graph is now acyclic, and 3) the precedence links indicate directed connections among the nodes.

A.4.1 From SDFG to DAPG — A Case

In order to illustrate our DAPG algorithm, we now transform the SDFG of a 2nd order IIR filter section into its corresponding DAPG. The initial DSP algorithm, the 2nd order difference equation, is given in equation A.12.

\[
y[n] = a_1 y[n - 1] + a_2 y[n - 2] + b_0 x[n] + b_1 x[n - 1] + b_2 x[n - 2]
\]  

(A.12)

From the difference equation, the SDFG is constructed (manually). In this example we apply the well-known direct form II structure (see e.g., [53]) as shown in figure A.6.

![Diagram of DAPG](image)

Figure A.6: The synchronous data flow graph for a second order filter section represented in a direct form II structure.

Recognize, that all nodes as well as all arcs have been assigned an arbitrary number — any node/arc number combination will lead to the same DAPG. In the SDFG, the nodes 2, 6, and 9 represent the multiplications \( b_i \) and the nodes 5 and 8 are the \( a_i \) multiplications. The four additions in the algorithm are shown in the graph as the nodes 1, 3, 4 and 7. The input from the ADC \( x[n] \), which is applied to node 1, is disregarded as discussed in section A.2. Similarly, the output to the DAC \( y[n] \), which is calculated at node 3, has been removed from the graph. Also, it should be noted, that we have inserted three extra nodes, 10, 11, and 12. The purpose of these nodes is to distribute a single input token to several other nodes which need this token as an input. Such nodes are known as fork nodes which theoretically have execution times equal to zero. Finally, the two delays (denoted \( z^{-1} \) in the signal flow notation) are represented in the graph with an "\( d \)" on arc 5 and 11, respectively. The interpretation is an initial token on each of these arcs. The \( b(0) \) vector therefore contains only zeros except for element 5 and 11 which both equals 1.
APPENDIX A. PREPROCESSING THE DSP ALGORITHMS

In this filter SDFG, all nodes produce and consume exactly one token on every arc. This is also indicated in figure A.6 with a "1" on each node's in- and output. As previously mentioned, such a graph is defined as a homogeneous SDFG. From the SDFG shown in figure A.6, the corresponding topology matrix is next constructed as shown in equation A.13.

\[ \Gamma = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix} \] (A.13)

The primary condition for the existence of a parallel schedule is that a PASS can be found, and this requires that \( \text{rank}(\Gamma) = s - 1 \); in this case 11, since the SDFG has 12 nodes. It is easily verified that the rank-condition holds for the actual topology matrix. Furthermore, the smallest integer vector in the null-space of \( \Gamma \) is a vector with only ones — and hence each node has to be executed once and only once in every sample period. This also confirms with our basic knowledge of digital filter structures.

Since all nodes are internal nodes, none of them can be invoked before a sufficient amount of tokens are available on the arcs. Initial tokens are provided on the inputs of the fork nodes 11 and 12, which are thus immediately executable. They produce tokens for the nodes 5, 6, 8, and 9 which can next be executed, and so on. Our PASS algorithm detects the following sequential schedule.

\[ \phi = \{11, 12, 5, 6, 8, 9, 4, 7, 1, 10, 2, 3\} \] (A.14)

The reason why this particular schedule is generated is due to the fact that our algorithm works on a list \( L \) which holds the nodes in a numerical order. For an example, two nodes which are simultaneously executable will therefore appear in \( \phi \) in a numerical order.

In the DAPG-algorithm, the nodes are assigned to the DAP graph in the order specified by the PASS. For each node assignment, the precedence relations are detected and precedence links are established from the node (which is currently being assigned) to a number of predecessor instances sufficient for providing the amount of tokens required for its \( k^{th} \) run. Since the SDFG is homogeneous, only one instance of each node is represented in the DAPG. The actual DAPG corresponding to the 2nd order filter SDFG is shown in figure A.7.

From the DAPG is should be evident that such a graphic representation of the DSP-algorithm is favourable as an input for the scheduling strategies. Basically, the inter-node precedences are visualized, but also the algorithm's inherent parallelism is very efficiently exploited. From figure A.7 it can be easily seen that all four multiplications in the filter are independent and can thus be executed in parallel if sufficient hardware resources are available. The same result could probably also had been found analytically, but for DSP-algorithms with a complexity slightly higher than the 2nd order filter this is not realistic, and hence formal techniques like our PASS
and DAPG algorithms are indispensable. The preprocessing of the DSP-algorithm(s) is therefore very essential for the further success of mapping onto a given target architecture.

The preprocessing is not quite finish yet. We have previously indicated, that realistic scheduling requires a detailed knowledge of the underlying target architecture, i.e., the actual processor interconnection topology, synchronization, the execution time of the graph nodes, and IPC times. Some of these informations must be represented in the DAPG, e.g., an estimate of the node execution time is efficiently indicated by a weight factor, $w_i$, related to each node in the graph. Similarly, the amount of tokens transmitted between the nodes has to be indicated as weight factors on each arc.

As an example, assume that the DAPG from figure A.7 has to be scheduled onto an architecture, where multiplications as well as additions require 1 time unit and 1 token is transmitted on every arc. In this scenario, the final DAPG which is ready for scheduling, will look like illustrated in figure A.8.

It is essential to note that the node-execution times as well as the communication amounts in general will take on individual values due to the actual subalgorithms and the amount of tokens they interchange. Concerning the communications, however, the current version of our DAPG algorithm may lead to inaccuracy (as compared to the initial SDFG) in the case of a multiple sample rate graph. To realize this circumstance, regard the simple SDFG displayed in figure A.9.

The topology matrix corresponding to this SDFG is

$$
\Gamma = \{2 - 3\} \quad \text{(A.15)}
$$

and thus the invocation vector (in the null space of gamma) indicates that node 1 and node 2 should be executed 3, respectively, 2 times within every sample period.

Taking a step-by-step walk through the DAPG algorithm, the precedence graph generated is
Figure A.8: The DAPG extended with runtime as well as communication loads. This graph holds the necessary information required concerning precedences, runtimes and IPC.

Figure A.9: A two-node SDFG characterized as being a multi sample rate system. shown in figure A.10.

Figure A.10: The generated DAPG. The subscript indicates the order of node assignment to the precedence graph.

It turns out that node 1 has to be assigned twice (1₁ and 1₂) to the DAPG before the first instance of node 2 (2₁) can be represented. Since node 2 requires 3 tokens upon execution,
precedence links to both instances of node 1 are assigned to the first run of node 2. Note however, that we do not specify the actual number of tokens passed from 1_1, respectively 1_2 to 2_1. The next "runnable" node is 1_3, and finally 2_2 is assigned to the DAPG. When the DAPG-algorithm is assigning precedence links to 2_2, all three instances of node 1 are made predecessors. Since this is obviously not necessary (taking into account the amount of tokens required by 2_2) the question is why? As slightly indicated, we do not know which one of 1_1 and 1_2 that provide 2, respectively 1 token (remember that three tokens are required in order to execute node 2). It is obvious that 2_2 must have a precedence link to 1_3 but due to the unknown token transfer from 1_1 and 1_2 to 2_1, 2_2 has to be dependent on both 1_1 and 1_2 as well as on 1_3.

This means that in the DAPG there is a uncertainty concerning the amount of tokens transferred on the precedence links. When preparing for realistic scheduling of the DAPG (by assigning communication loads the precedence links), we therefore have to refer back to the SDFG. In this particular case it says that 2 tokens are produced onto the arc on every invocation of node 1. The communication load on every precedence link is thus assigned 2 tokens — although we know that this is an overkill.

In order to overcome the problem, refinements must be made to the current version of the DAPG algorithm. Basically, the algorithm should be extended with a mechanism which “on-the-fly” assigns the communication loads simultaneously with the precedence links. This, however, will not be discussed here.
Appendix B

Randomly Generated DAP Graphs

Since we are focusing on DSP types of algorithms, the Synchronous Data Flow notation has been conveniently chosen for the graphic representation needed prior to the scheduling. In order to evaluate our scheduling strategies, a variety of DSP-algorithms (represented as SDF graphs) with different characteristics in terms of, e.g., 1) inherent parallelism, 2) computation/communication ratio, and 3) topology are required as test vehicles. Off-the-shelf DSP algorithms may be applied, but we would also like to expose our techniques with graphs having some more controlled characteristics. The heuristics will therefore (initially) be tested by randomly generated graphs. By “random” we mean graphs that are constructed randomly within some well-specified upper and lower bounds on several essential graph parameters.

In section A.4 we described the transformation of an SDF graph into a DAPG. This transformation is required before the scheduling can be initiated, primarily due to the facts that 1) SDF graphs may be cyclic, and 2) it may not be obvious from the SDF graph how many times the individual nodes actually should be invoked in order to maintain consistent token production/consumption among successive sample periods (multi sample rate graphs). Since parameters like inherent parallelism, regularity, computation- and communication loads are essential, randomly generated SDFGs can hardly serve the purpose as test vectors — primary because these parameters do not expose themselves clearly in the SDFG notation. The DAPGs, on the other hand, can be easily evaluated and generated according to these parameters. We therefore have selected the random DAPGs as representatives for the DSP algorithms during the initial test of our heuristics.

In this appendix, we therefore discuss the construction of a Graph Generator Tool (GGT) required for the test graph creation.

B.1 Specification of Parameters within the GGT

In a multiprocessor target architecture, we simply define the parallelism as the number of processors. It is obvious, that a DAPG which is very sequential in nature may not utilize optimally the parallelism provided by such an architecture. On the other hand, it is reasonable to believe that a graph with substantial inherent parallelism can be mapped onto the architecture in such a way that the hardware utilization becomes relatively high. For that reason, we want the GGT to have adjustable parameters which may be used to control the graph parallelism.

Before proceeding, we therefore need to introduce an expression for the graph parallelism. We rely on the definition given in [42]. Basically, this definition yields;
\[ \text{PAR} \equiv \left\lfloor \frac{t_{\text{SEQ}}}{t_{\text{CP}}} \right\rfloor \] (B.1)

where \( t_{\text{SEQ}} \) is the total sequential runtime of the algorithm and \( t_{\text{CP}} \) represents the runtime of the critical path in the DAPG.

From this definition it follows, that a completely sequential algorithm (or DAPG) has a degree of parallelism which equals 1, whereas an algorithm which is not strictly sequential has a degree of parallelism which is greater than 1.

In order to generate DAPGs with different degrees of parallelism, the graph topology therefore need to be adjustable. This means that GGT must be able to influence the interconnection among the graph nodes. We implement this facility by making 1) the number of successors (or predecessors), and 2) the precedence link generation variable.

Basically, a graph is build by first specifying the total number of nodes. Then, a hierarchy of levels containing different numbers of nodes within specified limits is constructed. Among nodes in different levels, precedence links are next inserted according to MIN/MAX limits.

When the graph topology has been created, node- and arc weights representing run times and IPC overheads are assigned randomly.

From the above, we now have the following specification of parameters required in generation of random DAPGs.

- Total number of nodes in the graph
- MIN/MAX number of nodes in the levels
- Selection among successor/predecessor for interconnection
- MIN/MAX number of successors/predecessor to each node
- Probability function for selection of successors/predecessors
- MIN/MAX node weight
- MIN/MAX arc weight

Limited by the MIN/MAX bounds, the various parameters will be selected randomly according to a uniform distribution. A number of graphs generated with the same set of parameters will therefore provide a test basis for a specific graph type.

It was briefly mentioned that the selection of successor/predecessor is due to a specific probability function. This probability function and the exact graph generation procedure are now introduced.

### B.2 Algorithms for Graph Generation

In this section we will discuss in some detail the algorithms developed for GGT. Initially, actual values for the parameters are specified. Based on the total number of nodes and the MIN/MAX values for the number of nodes in each level, a hierarchy is next constructed. This is illustrated in algorithm 19.

The primary idea behind this initial level hierarchy is that graph parallelism can be adjusted by allowing more or less nodes to reside in the same level. The reason being that precedence
links next have to be inserted between the nodes residing in different levels. Therefore, if we allow only a few nodes (relative to the total number of nodes) to be assigned to the same level, the graph parallelism is likely to be lower as compared to the situation where a larger number of nodes is assigned to the individual levels. In the former case, the total number of levels will be higher than in the latter. However, the way precedence links are inserted between the nodes has a major impact on the final graph topology. We therefore designed two strategies for link assignment which are described below.

### B.2.1 Linear Decreasing Probability

As pointed out, precedence links have to be assigned among nodes residing in different levels. The MIN/MAX number of successors/predecessors specify how many precedence links there should be assigned to each node. For simplicity, we will discuss assignment of links to successor nodes only — the strategy for assigning links to predecessor nodes is very similar. It should be emphasized, however, that link assignment is done to either successors or predecessors.

The starting point (using successors) is the top most level, i.e. level 1. For every node residing in this level, we have to select a number of successor nodes residing in levels below (i.e. levels with identification number > 1). The actual number of successors to the individual nodes is determined randomly from a uniform distribution in the interval specified by MIN/MAX successors. Initially, we assume that all nodes residing in levels below level 1 (henceforth the current level) have equal probability for being selected as successor to the node which is currently investigated. This means that nodes from the top-most levels are likely to have successors residing in levels near the bottom. As more and more nodes are assigned successors, we approach the bottom level, and thus still fewer nodes are available as successors which means that the “concentration” of precedence links becomes higher. In other words; as we are approaching towards the bottom of the DAP graph, the nodes will have more input arcs as compared to nodes residing in the top-to-middle part of the graph. Graphs constructed with equal connection probability for all potential successors therefore have a higher degree of connectivity at the bottom as compared to the top.

In order to alleviate this problem, we therefore introduce a linear decreasing probability function applied for selecting successor nodes. The idea is illustrated in figure B.1.

Figure B.1 shows the probabilities $P(j)$ for selecting nodes from levels below the current level. We denote the total number of levels below the current one by $x$. Since $x$ is known and since it must be true that $\sum_{j=1}^{x} P(j) = 1$, an expression for $P(j)$ for $j = 1, 2, ..., x$ can be found. Such an expression is devised on the basis of the following set of equations.

1: $P(j) = \alpha \cdot j + \beta$

2: $P(j) = 0$ for $j = x + 1 \Rightarrow \beta = -\alpha \cdot (x + 1)$

#### Algorithm 19: The algorithm applied for building level hierarchy. The top most level has number 1.

```plaintext
level_count = 1;
While there are still nodes not assigned to a level;
  Select an arbitrary number of nodes (act_num) to be assigned to level level_count;
  If act_num > number of remaining nodes
    Then act_num = number of remaining nodes;
  Assign act_num nodes to level level_count;
End;
level_count + +;
```
3: \( \alpha = P(2) - P(1) \) and \( P(1) > P(2) > 0 \) for \( x \geq 2 \)

Now, for \( x = 2 \) (i.e., for \( j = 1 \) and \( j = 2 \)), the following set of equations can be specified.

\[
\begin{align*}
P(1) & = \alpha + \beta \\
P(2) & = 2\alpha + \beta
\end{align*}
\] (B.2) (B.3)

Substituting into these equations the expression for \( \beta \), the following two equations are obtained:

\[
\begin{align*}
P(1) & = \alpha - \alpha \cdot (x + 1) = -\alpha \cdot x \\
P(2) & = 2 \cdot \alpha + (-\alpha \cdot (x + 1)) = \alpha \cdot (1 - x)
\end{align*}
\] (B.4) (B.5)

In order to eliminate \( \alpha \), we are now utilizing that \( \alpha = P(2) - P(1) \), which leads to

\[
\begin{align*}
P(1) & = \frac{x}{x - 1} \cdot P(2) \\
P(2) & = (1 - \frac{1}{x}) \cdot P(2)
\end{align*}
\] (B.6) (B.7)

These two equations are next applied in the equation \( \alpha = P(2) - P(1) \), and after a few manipulations, we obtain

\[
\alpha = -\frac{1}{x} \cdot P(1)
\] (B.8)

From this, we are also able to obtain an expression for \( \beta \):

\[
\beta = (1 + \frac{1}{x}) \cdot P(1)
\] (B.9)
Now, $\alpha$ and $\beta$ are described as functions of the number of remaining levels, $x$. It is therefore straightforward also to get a general expression for the probabilities $P(j)$.

$$P(j) = \frac{-1}{x} \cdot P(1) \cdot j + (1 + \frac{1}{x}) \cdot P(1)$$  \hspace{1cm} (B.10)

$$\Downarrow$$

$$P(j) = P(1) \cdot \left( \frac{x + 1 - j}{x} \right)$$  \hspace{1cm} (B.11)

However, the value of $P(1)$ is also a function of $x$, but initially we for simplicity choose $P(1) = 1$ in order to calculate $P(j)$ for $j = 2, 3, \ldots, x$. Afterwards, we scale all values ($P(j)$ for $j = 1, 2, \ldots, x$) by a factor $s$, which is determined according to the equation

$$s = \frac{1}{\sum_{j=1}^{x} P(j)}$$  \hspace{1cm} (B.12)

Finally, the calculated probabilities are applied in order to select appropriate levels below the current one. Our algorithm is outlined in algorithm 20.

\begin{algorithm}
Generate an arbitrary number $\beta$ in the interval $[0;1]$;
For all levels $j$ below the current one, starting with $j = 1$;
Calculate the probability interval $\eta$ based on the level probabilities $P(j)$;
If $\beta \in \eta$ then select level $j$;
End;
\end{algorithm}

Algorithm 20 : The algorithm employed for level selection. Note that the random number initially generated ($\beta$) is a stochastic variable with uniform probability distribution in the interval $[0;1]$. This particular circumstance is what makes it possible to select the successor level based on the precalculated values $P(j)$.

By using this link assignment technique, we eliminate the majority of connections from nodes in the top-most levels to nodes residing in levels near the bottom. However, from equation B.11 it is evident that for a large number of levels ($x$) below the current one, there exists only a minor probability variation among adjacent levels. Therefore, the linear decreasing probability function may not be able to efficiently eliminate the high concentration of precedence links at the bottom of the graph, and thus we suggest an alternative probability function.

B.2.2 Reciprocal Decreasing Probability

The alternative probability function is based on the reciprocal function and is basically given as

$$P(j) = \frac{1}{j} \quad j = 1, 2, \ldots, x$$  \hspace{1cm} (B.13)

where $x$ still denotes the total number of levels below the current one.

In order to prepare for level selection according to the procedure described in algorithm 20, all the probabilities are scaled by the factor

$$s = \frac{1}{\sum_{k=1}^{x} \frac{1}{k}}$$  \hspace{1cm} (B.14)

so that the final probabilities are given by
B.3 Representing the DAP Graphs

The procedure described above outlines the method applied in order to generate random DAPGs. Remember that our scheduling framework requires as input DSP algorithms in terms of SDFGs. Initially, one may then think that the DAPGs have to be converted back to SDFGs before they are applicable as input for scheduling. This, however, is impossible since our DAP-to-SDF algorithm is a non-invertible function due to its stochastic nature imposed by the arbitrary node list $L$ and due to the fact that precedence link assignment do not take into account the amount of tokens transferred between the various node instances in the DAPG — see algorithm 18.

Fortunately, the SDF notation allows representation of DAPGs. This is possible because we do not bother about the parent SDFG to a randomly generated DAPG. Because of that, the DAPG can be regarded as an acyclic homogeneous SDFG, i.e. an executable (error free) SDFG with $b = 0$ and $q_{\text{min}} = 1$.

Based on this observation, the GGT assigns to every node and arc (in the DAPG) identification numbers as well as arbitrary arc weights which are next employed to set up a topology matrix $\Gamma$ — note that this matrix in every row contains two numerically equal numbers with different sign. Furthermore, from the generated DAPG, GGT detects predecessor relationships for all nodes — those which have no predecessors are classified as external nodes, all others as internal nodes (in this work we also use the term root nodes for external nodes and internal nodes which have no successor nodes are defined as leaf nodes). This node information is kept along with the arbitrary node weights also assigned by the GGT.

Summarizing, our GGT provides DAPGs generated arbitrarily within some user-specified MIN/MAX bounds and represented in terms of acyclic SDF graphs.

B.4 Defining the Randomly Generated Test Graphs

As we did mention in the introduction to this appendix, our scheduling heuristics should be tested on DSP benchmarks as well as on randomly generated DAP graphs. In this section we therefore define the various randomly generated graphs for test purposes.

Basically, we divide our test sets into three major parts according to the number of nodes in the graphs. We generate graphs with 25, 50 and 100 nodes, respectively. We consider these graph sizes to be representative for most DSP algorithms. Next, within each of these categories, we selected three different degrees of parallelisms as defined by equation B.1. In order to represent a broad range of different algorithm topologies (from the almost sequential to the highly parallel) we generate graphs within the parallelism intervals $i)$ $2 - 4$, $ii)$ $6 - 8$, and $iii)$ $10 - 12$.

Now, concerning the node execution times, all graphs contain nodes with run times uniformly distributed between 4 and 20, i.e., $w_i \in [4; 20]$. Similarly, we restrict the number of tokens passed between interconnected nodes to be uniformly distributed between 1 and 5, meaning that $c_{ij} \in [1; 5]$ ($c_{ij}$ being the variable which specifies the amount of tokens transmitted from $n_i$ to $n_j$). We note that the mean computation time is $\mu_{\text{comp}} = 12$ and the mean number of communicated tokens is $\mu_{\text{comm}} = 3$. By adjusting the communication time for a single token, we can now easily control the ratio between the mean communication and mean computation time.

According to different 1) graph size and 2) parallelism, we have defined 9 graph sets as shown in table B.1. From this table we note that the parameter “MIN/MAX nodes in the levels” (the
Table B.1: Nine sets each containing five DAP graphs constitute our library of randomly test graphs which are all generated using “reciprocal decreasing probability”.

<table>
<thead>
<tr>
<th>Set #</th>
<th>No. of nodes</th>
<th>Parallelism</th>
<th>MIN/MAX nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>2-4</td>
<td>1/1</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>6-8</td>
<td>6/10</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>10-12</td>
<td>12/14</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>2-4</td>
<td>1/1</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>6-8</td>
<td>6/10</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>10-12</td>
<td>12-14</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2-4</td>
<td>1/1</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>6-8</td>
<td>4/6</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>10-12</td>
<td>6/10</td>
</tr>
</tbody>
</table>

right most column) is used to control the actual degree of parallelism. For graphs having a low parallelism, only 1 node resides in every level, whereas for highly parallel graphs, the number of nodes in every level is increased substantially.
Appendix C

Published Papers

Throughout the preparation of this thesis, four papers were published at internationally recognized conferences in order to verify the firm substance of our ideas. In the following, we cite these papers which, upon request, are available from the author.

C.1 Paper 1

Title: On Realistic and Efficient Scheduling of DSP Algorithms onto Multiprocessor Architectures.

Authors: Peter Koch, Kallol Bagchi, and Kjeld Hermansen.

Presented at: The 26th Asilomar Conference on Signals, Systems & Computers, Pacific Grove, California, USA, October 1992, Sponsored by the IEEE Computer Society and the ACM.

Presented by: Peter Koch as a speech.

Published in: The Conference Proceedings, pp. 651-655, ISBN 0 8186-3160 0.

C.2 Paper 2

Title: Implementation Studies of Efficient and Realistic Multi Signal Processor Solutions for DSP Applications.

Authors: Peter Koch, Kallol Bagchi, and Kjeld Hermansen.


Presented by: Peter Koch as a speech.

Published in: The Conference Proceedings, pp. 7.3.1-7.3.10, ISBN 0 7008 0480 3.
C.3 Paper 3

Title: *Static Scheduling of Data Independent Scientific Computations onto Multi-DSP Architectures using Simulated Annealing.*

Authors: Peter Koch and Hans Jensen.


Presented by: Peter Koch as a poster


C.4 Paper 4

Title: *Simulated Annealing Applied to Compile Time Scheduling onto Multiple DSP Architectures*

Authors: Hans Jensen and Peter Koch

Presented at: *The 7th International Conference on Parallel and Distributed Computing and Systems, Washington, D.C., USA, October, 1995, Sponsored by IASTED/ISMM*  

Presented by: Peter Koch as a speech

Published in: *The Conference Proceedings, pp. 85-88, ISBN 0 88986 228 1*