

Architecture Synthesis for Adaptive Multiprocessor Systems on Chip

Dissertation
zur Erlangung des akademischen Grades
“doctor rerum naturalium”
(Dr. rer. nat.)
in der Wissenschaftsdisziplin “Technische Informatik”

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

Von
Harold Ishebabi

Potsdam, den 29.05.2009

1. Berichterstatter: Prof. Dr. rer. nat. Christophe Bobda
2. Berichterstatter: Prof. Paul Chow
3. Berichterstatter: Prof. Dr. Achim Rettberg

This work is licensed under a Creative Commons License:
Attribution - Noncommercial - Share Alike 3.0 Unported
To view a copy of this license visit
<http://creativecommons.org/licenses/by-nc-sa/3.0/>

Published online at the
Institutional Repository of the University of Potsdam:
URL <http://opus.kobv.de/ubp/volltexte/2010/4131/>
URN <urn:nbn:de:kobv:517-opus-41316>
<http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-41316>

Hiermit erkläre ich, dass die Arbeit an keiner anderen Hochschule eingereicht sowie selbständig und nur mit den angegebenen Mitteln angefertigt wurde.

Potsdam, 29.05.2009

(Harold Ishebabi)

Abstract

This thesis presents methods for automated synthesis of flexible chip multiprocessor systems from parallel programs targeted at FPGAs to exploit both task-level parallelism and architecture customization. Automated synthesis is necessitated by the complexity of the design space. A detailed description of the design space is provided in order to determine which parameters should be modeled to facilitate automated synthesis by optimizing a cost function, the emphasis being placed on inclusive modeling of parameters from application, architectural and physical subspaces, as well as their joint coverage in order to avoid pre-constraining the design space. Given a parallel program and a set of an IP library, the automated synthesis problem is to simultaneously (i) select processors (ii) map and schedule tasks to them, and (iii) select one or several networks for inter-task communications such that design constraints and optimization objectives are met. The research objective in this thesis is to find a suitable model for automated synthesis, and to evaluate methods of using the model for architectural optimizations. Our contributions are a holistic approach for the design of such systems, corresponding models to facilitate automated synthesis, evaluation of optimization methods using state of the art integer linear and answer set programming, as well as the development of synthesis heuristics to solve runtime challenges.

Acknowledgement

This thesis results from my research work at the Professorship for Computer Engineering of the Institute of Computer Science of the University of Potsdam from 2007 to 2009. Many people have contributed to this work or assisted me in one way or the other. While I cannot provide a full list in this short acknowledgement, I would like to explicitly thank a few individuals.

Especially I would like to thank my advisor Prof. Dr. rer. nat. Christophe Bobda for the opportunity to work in the emerging field of flexible Chip Multiprocessor Systems at his Professorship, for productive discussions at the Technical University of Kaiserslautern that ultimately led to this work, and for his leadership throughout my work.

Many thanks to Prof. Paul Chow and Prof. Dr. Aachim Rettberg for their interest in my thesis and for expending their invaluable time to review my work, and to serve on my thesis committee.

I would like to express my gratitude to Prof. Dr. Torsten Schaub, Dipl.-Inform. Martin Gebser, Dipl.-Inform. Roland Kaminski and Dipl.-Inform. Benjamin Kaufmann all at the Professorship for Knowledge Processing and Information Systems for their support on Potassco suite for Answer Set Programming, to colleagues Dipl.-Ing. Philipp Mahr for coordinating development work on the platform PinHat particularly when I was working from a great distance, to Manuela Zeitner for general organizational assistance, and to working students Dipl.-Inf. Christian Lörchner and Oliver Matheis for their collaboration on PinHat. Thanks also to Dr. Humphrey Rutagemwa for reviewing this thesis in the early stages, and to Dr. Kizito Ssamula Mukasa for guiding me in many ways.

To my loving wife Alice thank you for your patience and support. To my parents and siblings, as well as to my friends and former colleagues Dr. Konstantinos Nikitopoulos, Msc. Venkatesh Ramakrishnan, Neema Mukasa, Vili Mulla, Ramde Mamadou, Raimund Doerr, Deogratias Kironde and Elisabeth Boettcher, thank you so much for your moral support. Finally to my colleagues at Spectrum Signal Processing: thank you for bearing with me.

Contents

| | |
|--|----------|
| List of Figures | 1 |
| List of Tables | 3 |
| List of Algorithms | 5 |
| 1 Introduction | 1 |
| 1.1 Design Phases for Flexible CMP Systems | 3 |
| 1.2 Research Objective | 4 |
| 1.2.1 Architecture Optimality | 4 |
| 1.2.2 Efficiency of a Design Flow | 5 |
| 1.3 Main Contributions | 5 |
| 1.4 Organization | 6 |
| 2 Design Space of CMP Systems | 7 |
| 2.1 System Architecture Subspace | 7 |
| 2.1.1 Peripherals | 7 |
| 2.1.2 Processing Elements | 7 |
| 2.1.3 Communication Infrastructure | 10 |
| 2.1.4 Memory Architecture | 11 |
| 2.2 Application Subspace | 12 |
| 2.2.1 Task Mapping | 12 |
| 2.2.2 Scheduling | 12 |
| 2.2.3 Network-Services | 14 |
| 2.2.4 Reconfigurability | 18 |
| 2.2.5 Programming Paradigm | 18 |
| 2.3 Physical Subspace | 18 |
| 2.4 Design Space Exploration | 19 |
| 2.5 Related Work | 21 |
| 2.5.1 Processor Design and Exploration | 21 |
| 2.5.2 NoC Design and Exploration | 22 |
| 2.5.3 CMP Design Tools and Methodologies | 22 |

| | | |
|----------|--|-----------|
| 2.5.4 | Mapping Applications to Fixed CMP | 24 |
| 2.5.5 | Domain-Specific Exploration | 24 |
| 2.6 | Chapter Summary | 24 |
| 3 | Problem Formulation | 27 |
| 3.1 | Proposed Design Flow for Flexible CMP Systems | 27 |
| 3.2 | ILP Formulation | 29 |
| 3.2.1 | Basic Formulation | 29 |
| 3.2.2 | Scheduling | 52 |
| 3.2.3 | Makespan | 63 |
| 3.3 | Chapter Summary | 68 |
| 4 | SAT-Based Synthesis | 73 |
| 4.1 | Methods for Solving SAT Problems | 73 |
| 4.1.1 | The Resolution Proof System and the DP Algorithm | 73 |
| 4.1.2 | The DPLL Algorithm | 74 |
| 4.2 | Answer Set Programming and the Potassco Suite | 77 |
| 4.3 | Encoding the Synthesis Problem as ASP Programs | 78 |
| 4.3.1 | Task Mapping | 78 |
| 4.3.2 | Processor Sharing | 78 |
| 4.3.3 | Processor Area Constraints | 78 |
| 4.3.4 | Network Usage Constraints | 79 |
| 4.3.5 | Network Capacity and Area Constraints | 79 |
| 4.3.6 | Scheduling Constraints | 79 |
| 4.3.7 | Makespan | 80 |
| 4.3.8 | Objective Function | 80 |
| 4.4 | Comparison of Synthesis Results | 81 |
| 4.4.1 | Non-Realtime Applications | 81 |
| 4.4.2 | Realtime Preemptive Scheduling | 84 |
| 4.4.3 | Makespan Optimization | 89 |
| 4.5 | Chapter Summary | 90 |
| 5 | Greedy-Like Heuristics | 93 |
| 5.1 | Substructure of the Synthesis Problem | 94 |
| 5.1.1 | Instances Without Resource Constraints | 95 |
| 5.1.2 | Practical Challenges and Solutions | 98 |
| 5.1.3 | Instances with Resource Constraints | 99 |
| 5.2 | Replace and Search Heuristic | 99 |
| 5.2.1 | Processor Assignment | 101 |
| 5.2.2 | Network Assignment | 101 |
| 5.2.3 | Optimization Loop | 104 |

| | | |
|----------|---|------------|
| 5.2.4 | Area Recovery | 106 |
| 5.3 | Group Growing Heuristic | 108 |
| 5.4 | Priority Assignment Heuristic | 108 |
| 5.5 | Comparison of Synthesis Results | 111 |
| 5.5.1 | Non-Realtime Applications | 111 |
| 5.5.2 | Realtime Preemptive Scheduling | 114 |
| 5.5.3 | Makespan Optimization | 114 |
| 5.6 | Chapter Summary | 121 |
| 6 | Conclusion | 123 |
| 6.1 | Summary | 123 |
| 6.2 | Outlook | 124 |
| A | Glossary and Notation | 125 |
| A.1 | Glossary | 125 |
| A.2 | Notation | 126 |
| B | The Synthesis Tool | 129 |
| | Bibliography | 135 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Software flexibility vs performance of processing elements | 2 |
| 1.2 | Determination of system architecture | 3 |
| 1.3 | A high-level implementation flow for flexible CMP systems on FPGA | 5 |
| 2.1 | CMP design space | 8 |
| 2.2 | Network Switching Techniques | 15 |
| 2.3 | Setup for extraction of network parameters | 17 |
| 2.4 | Y-Shaped design space exploration | 19 |
| 2.5 | Methods for exploring the design space | 20 |
| 2.6 | Illustration of exploration methods (source[1]) | 21 |
| 3.1 | Proposed architecture synthesis flow | 28 |
| 3.2 | Processor core showing inclusion of network interfaces | 34 |
| 3.3 | Application topologies | 40 |
| 3.4 | Extracting application traffic from simulations | 40 |
| 3.5 | Continued on the next page | 44 |
| 3.5 | Estimated task run time on MB for different number of tasks | 45 |
| 3.6 | Solver runtime versus number of processors available | 46 |
| 3.7 | Solver runtime versus number of networks available | 46 |
| 3.8 | Solver runtime versus number of tasks | 47 |
| 3.9 | Gaps of found combinatorial solutions over relaxed solutions | 47 |
| 3.10 | Synthesized architectures under scheduling extension | 62 |
| 3.11 | WLAN Program Graph | 68 |
| 4.1 | A decision graph for the DPLL algorithm | 75 |
| 4.2 | Implication graph for the foregoing example | 76 |
| 4.3 | Continued on next page | 82 |
| 4.3 | ASP-ILP Comparison : Increasing the number of processors | 83 |
| 4.4 | Continued on next page | 85 |
| 4.4 | ASP-ILP Comparison : Increasing the number of networks | 86 |
| 4.5 | Continued on next page | 87 |
| 4.5 | ASP-ILP Comparison : Increasing the number of tasks | 88 |

| | | |
|-----|---|-----|
| 4.6 | Synthesized architectures under preemptive scheduling | 90 |
| 5.1 | Overall structure of synthesis heuristics | 100 |
| 5.2 | Moving tasks eliminates edges and exposes new ones | 104 |
| 5.3 | Group growing | 111 |
| 5.4 | Continued on next page | 112 |
| 5.4 | Greedy-ILP comparison : Increasing the number of processors | 113 |
| 5.5 | Continued on next page | 115 |
| 5.5 | Greedy-ILP comparison : Increasing the number of networks | 116 |
| 5.6 | Continued on next page | 117 |
| 5.6 | Greedy-ILP comparison : Increasing the number of tasks | 118 |
| 5.7 | Continued on next page | 119 |
| 5.7 | Greedy-ILP Comparison : Increasing the number of tasks, objective value | 120 |

List of Tables

| | | |
|------|--|-----|
| 3.1 | Number of constraints | 38 |
| 3.2 | Number of decision variables | 38 |
| 3.3 | MPI applications in the experiment | 38 |
| 3.4 | Parameters of used processors | 40 |
| 3.5 | Parameters of used networks | 41 |
| 3.6 | A summary of application characteristics | 43 |
| 3.7 | Integer solutions | 45 |
| 3.8 | Synthesized architectures for non-realtime parallel programs | 48 |
| 3.9 | Truth table for inequalities (3.42) and (3.43) | 54 |
| 3.10 | WLAN tasks | 55 |
| 3.11 | WCDMA tasks | 56 |
| 3.12 | Traffic pattern for WLAN | 56 |
| 3.13 | Traffic pattern for WCDMA | 57 |
| 3.14 | Parameters of used processors | 57 |
| 3.15 | Synthesis results under scheduling extension | 61 |
| 3.16 | Makespan path lengths for WLAN | 69 |
| 3.17 | Synthesis results for makespan minimization | 70 |
| 4.1 | ASP-ILP comparison under Scheduling | 89 |
| 4.2 | ASP-ILP comparison for makespan optimizations | 90 |
| 5.1 | Greedy-ILP comparison under scheduling | 118 |
| 5.2 | Greedy-ILP comparison for makespan optimization | 120 |
| B.1 | Supported Kernel IDs | 132 |

List of Algorithms

| | | |
|----|--|-----|
| 1 | Determining context switching cost using RMA | 57 |
| 2 | Deadline relaxation during RMA | 58 |
| 3 | Multi-pass optimization for makespan minimization | 63 |
| 4 | Determining context switching cost using RMA with makespan extension | 66 |
| 5 | Generic greedy Algorithm [2] | 97 |
| 6 | Heuristic synthesis main steps | 100 |
| 7 | Initial processor assignment | 101 |
| 8 | Allocating a processor for a group of tasks | 102 |
| 9 | Initial network assignment | 102 |
| 10 | Allocating a network resource for an edge | 103 |
| 11 | Assigning an edge to a network | 103 |
| 12 | Optimization loop using backtracking | 105 |
| 13 | Area optimization | 107 |
| 14 | Optimizing loop using group growing | 109 |
| 15 | Priority based assignment | 110 |

1 Introduction

Transistor densities in microchips have been doubling approximately every two years following a trend that was predicted by Gordon Moore in his 1965 paper, “Cramming more components onto integrated circuits” [3]. With smaller transistors, the clock frequency of microprocessors could be increased from one technology node to another during a period covering approximately four decades. This, coupled with design innovations in instruction-set architectures, memories and micro-architectures, has led to a dramatic increase in the performance of microprocessors.

Spurred by these technological advances, the complexity of softwares and algorithms has been steadily increasing. That in turn has been leading to a need for higher performances. Unfortunately, the semiconductor industry has reached a point where the performance cannot be further increased by simple transistor scaling because of three major challenges.

The first challenge is what is known as the power wall. Power consumption in microprocessors has two components: static and dynamic. Dynamic power consumption is a linear function of the clock frequency, therefore, increasing the frequency undesirably increases the consumption. Even though transistor shrinking allows the supply voltage to be scaled down and thereby reduce dynamic consumption, doing so increases the static component which is becoming a major source of power consumption for technologies below the 90nm node [4]. The reason is that, in order to compensate for slower transistor switching due to lower supply voltage, the threshold voltage needs to be lowered. Since the threshold voltage appears as a negative exponent in subthreshold current, compensating for performance increases subthreshold static power consumption. Moreover, transistor scaling has led to the reduction of gate oxide thickness, leading to increased static power consumption due to gate oxide leakage. However, recent introduction of high-*k* dielectrics such as hafnium dioxide is significantly reducing the impact of this last static component.

The second challenge is the memory wall. Memory latency is a bottleneck in computer architectures [5] since access times have not been able to keep pace with clock frequencies, causing a wastage of clock cycles. This problem is particularly severe with off-chip memories, where wire delays are significantly larger compared to gate delays.

The third challenge is caused by diminishing returns from aggressive exploitation of instruction level parallelism. Significant area and power overhead are not justified by small performance improvements resulting from the detection and exploitation of the parallelism on processors [5]. Diminishing returns are also experienced with Very Large Instruction Word (VLIW) architectures [5] because as the number of slots is increased, the size of multiplexors serving the read/write ports of the register file grows more than linearly [5, 6]. Even though register-file partitioning reduces this problem, doing so limits opportunities for instruction-level parallelism because of additional clock cycle latencies required to exchange data between slots. Techniques such as chaining of register file access requests lead to the same effect.

Two emerging solutions to these challenges are parallel and reconfigurable computing. Parallel computing with multiple processors has traditionally been in the high-performance computing domain, where supercomputers and computer clusters have been long in use, with early systems dating back to the early 1960s. Recently, multi-processor and multi-core systems have been introduced for per-

sonal and embedded computing to mitigate the three technological challenges. These architectures can lead to higher performance through the exploitation of task-level parallelism without incurring severe power penalties.

As opposed to multi-processor systems, multi-core systems have two or more processors on a single die. The latter systems are also known as Chip Multi-Processors (CMPs). In the embedded domain the terminology Multi-Processors System-On-Chip (MPSoC) is used to characterize heterogeneous systems consisting of general and special purpose processors as well as reconfigurable components and Application-Specific Integrated Circuits (ASICs) on a single die. CMP systems can be perceived as a subset of MPSoC systems containing instruction-set components only, possibly with accelerators coupled to them.

In contrast to CMP systems, higher performance in reconfigurable computing is obtained through customization since that reduces (or eliminates) overhead associated with instruction scheduling and explicit data movements. Figure 1.1 shows a classification of processing elements according to software flexibility versus performance. At extreme ends we have General Purpose Processors (GPPs) which are fully software flexible, and ASICs which are not flexible. Even though all elements in the figure are essentially application-specific integrated circuits, the term ASIC is now widely used to refer to non-programmable and non-reconfigurable integrated circuits. Moving away from GPPs, Domain-specific Processors (DPs) and Application-Specific Instruction-set Processors (ASIPs) trade software flexibility for performance. Higher performance is obtained through specialization rather than through exploitation of high-level parallelism.

Reconfigurable devices, notably Field Programmable Gate Arrays (FPGAs), lead to an even higher performance because of full customization for specific applications. From the perspective of software programmability, FPGAs are less flexible devices compared to processors. However, because of their reconfigurable fabric, they are sometimes classified as being more flexible compared to ASIPs [7]. Between ASIPs and FPGAs, there is a class of reconfigurable ASIPs (rASIPs), which are partially reconfigurable ASIPs.

The concept of reconfigurable computing was introduced in 1959 by Gerald Estrin [8] to achieve gains in computational speed by exploiting special-purpose but reconfigurable hardware structures. Modern systems in reconfigurable computing follow the same idea by integrating one or several general purpose machines with one or several reconfigurable structures or devices. The capacity of modern FPGAs has led to a new class of architectures that can be viewed as multi-processor systems on FPGAs, targeting both single [9, 10] and multi-FPGA platforms [11, 12]. FPGA manufacturers such as Xilinx and Altera are supporting this field by providing ready to use components such as soft- and hard-core processors, bus-based communication infrastructures, and peripheral cores for off-chip

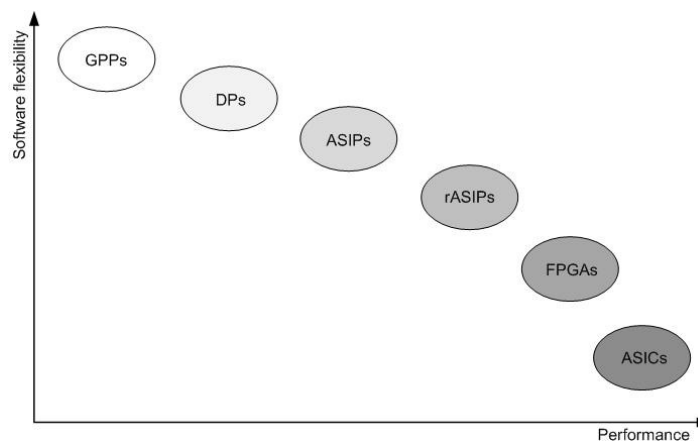


Figure 1.1: Software flexibility vs performance of processing elements

communications.

Such flexible CMP systems are appealing because they combine the benefits of parallel and reconfigurable computing: task-level parallelism can be exploited on customized systems. This possibility is particularly advantageous in the embedded domain, where the diversity of applications renders impossible the activity of finding a single CMP system that can satisfactorily meet the requirements of all applications that can be mapped to any given static architecture.

A second advantage of flexible CMP systems is adaptivity. Nikitovic and Brorsson proposed in 2002 a simple adaptive CMP system [13], where adaptivity is obtained by switching one of fixed embedded processor cores on and off to adjust the system to prevailing workload to save energy. Fully adaptive systems can be obtained by additionally changing the communication infrastructure and the functionality of hardware accelerators either at compile-time or at runtime through (partial) reconfiguration.

Flexible CMP systems can be employed in applications such as pattern matching, video streaming, distributed arithmetic, adaptive control, cryptography and software defined radios [7]. These systems are the focus of this thesis. The rest of this chapter discusses their design approaches and tools, as well as the objective of the research conducted as part of this thesis. The chapter concludes with an overview and the main contribution of this thesis.

1.1 Design Phases for Flexible CMP Systems

The task of implementing flexible CMP systems on FPGAs is very complex consisting of four phases. The first is the design of system components such as processors, memories and buses. In order to increase design productivity such components are usually made available as Intellectual Property (IP) cores, reducing this design activity to a selection problem. Knowledge about the application can be used in the selection to prune the design space. However, care must be exercised to avoid limiting options in later phases. IP-based design is assumed throughout this thesis.

In the second phase the system architecture is determined. This involves the selection of storage, I/O and processing elements, the mapping of computational tasks to processors, and the selection and allocation of communication resources, taking into consideration system and application constraints. Figure 1.2 depicts the input and the output of this design phase.

In the third phase, the generated system architecture is translated into a Register Transfer Level (RTL) description of the system. Additionally, the software application code is compiled for target processors, and appropriate drivers are configured for use. Finally, the RTL description is synthesized in the final phase to obtain the configuration bitstream using traditional FPGA synthesis flows.

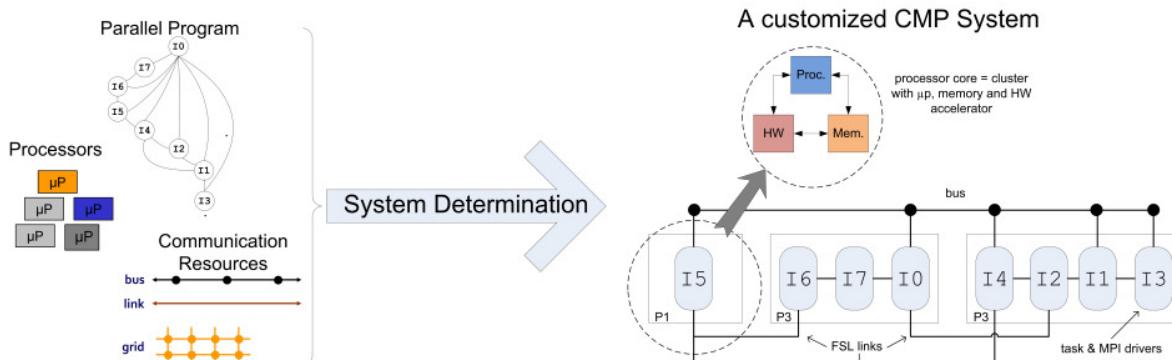


Figure 1.2: Determination of system architecture

1.2 Research Objective

Figure 1.3 summarizes the design flow for flexible CMP systems on FPGAs. A preliminary work towards this flow was started at the University of Kaiserslautern by the former research group “Self-Organizing Embedded Systems”. The result of the effort was a design tool called PinHat [9, 7] which covers the two lower design phases: System Integration and FPGA synthesis. The design approach and the tool-chain automates system integration based on IP reuse via abstraction, offering a path to vendor-specific FPGA synthesis tools. The contributions in this thesis stem from research effort to address design challenges arising from the second phase in order to complete the design flow. The main goal is to find an efficient way of determining optimum architectures from parallel programs. A prerequisite to this goal are definitions of (1) optimum architectures and (2) efficiency of corresponding design flows.

1.2.1 Architecture Optimality

A semiconductor design point often represents a trade-off between flexibility, area, energy efficiency, performance, and manufacturing objectives (yield and reliability). Consequently when more than one of these conflicting design goals are being simultaneously optimized, optimality is defined in pareto sense: generally, an optimum architecture is one that is located at the pareto front, implying that none of the five design objectives can be further improved without worsening the others. However, because the target platforms in this work are field programmable devices, only performance and energy efficiency are immediate optimization objectives: flexibility is inherent in FPGAs, area is a hard boolean constraint, and the field application nature means that manufacturing objectives have already been met. In this context, a hard boolean constraint is one whose fulfillment suffices, with no additional benefits beyond the mere fulfillment.

Static energy dissipation is caused by leakage currents from seven different sources in a CMOS device [14, 15], of which subthreshold and gate oxide leakages are dominant. At the CMP system level on FPGAs, the only method of optimizing for leakage is to reduce the size of the system so that some parts of the devices can be deactivated, or to actively manage system consumption through techniques such as powering-off components or active body-biasing. In FPGAs, the dynamic energy component is largely consumed in the interconnect fabric with the consequence that system level optimizations methods must be able to account for variations in routing algorithms. Therefore, generic energy dissipation models that capture characteristics of FPGA devices and routing algorithms need to be developed for CMP system level energy optimizations. That is an open and interesting area for future research. This thesis focuses on the remaining optimization objective: performance.

Within this context, performance is an umbrella term for one of the following:

1. The time needed to complete a certain set of tasks for a terminating program.
2. The time needed to complete a certain set of tasks for one period of a non-terminating program. Depending on application requirements, this can be used to optimize for processing latency or throughput: shortening the time reduces the latency, whereas increasing the time by way of task-level pipelining increases the throughput. Nevertheless, task-mapping must be such that the execution time of the most critical task as well as associated scheduling and communication overhead are minimized.

Consequently, in this thesis, an optimum architecture is the one that leads to the lowest execution time for a set I of given tasks in a parallel program while meeting device and system constraints. If \mathcal{J} is the set of all tasks in the parallel program, then $I = \mathcal{J}$ for case 1 above, and $I \subseteq \mathcal{J}$ for case 2.

This definition makes it possible to target different performance objectives without altering the basic optimization method. Because of the combinatorial nature of this design problem, the optimum

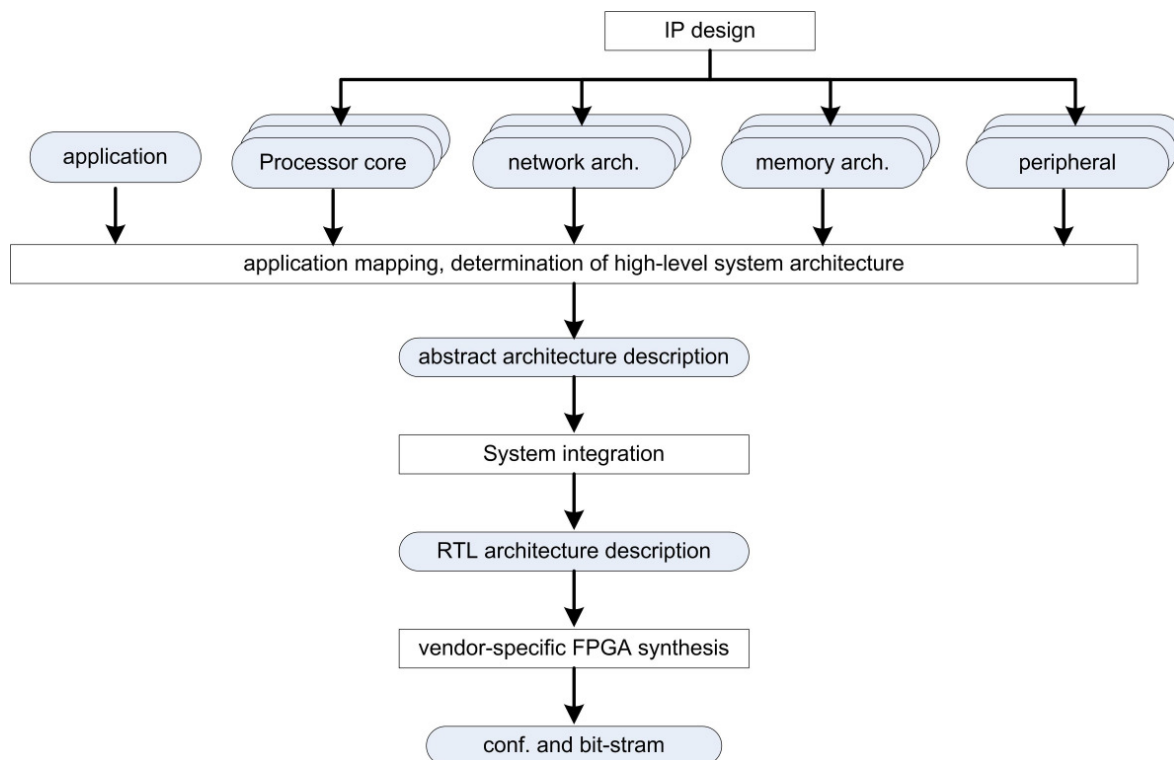


Figure 1.3: A high-level implementation flow for flexible CMP systems on FPGA

architecture may not be unique. Nevertheless, the intent is to find one such architecture (and not to enumerate all optimum architectures).

1.2.2 Efficiency of a Design Flow

Even though the term “efficient design flow” is widely used in the EDA industry and in the academia no metric has been publicized that can be used to measure the efficiency of a particular flow in absolute or relative terms. A flow is perceived as being efficient if it enables designers to reach their objectives easily and in reasonable time. These are arguably subjective characteristics, but still, they provide a guidance as to what constitutes an efficient flow, namely high abstraction and automation. That in turn implies the use of formal models that can unambiguously capture all necessary CMP design parameters to enable a rapid determination of application-specific optimum architectures, where optimality is defined in the sense described in the previous subsection.

1.3 Main Contributions

Thus, the research objective in this thesis is to find a suitable model for automated synthesis, and to evaluate methods of using the model for architectural optimizations. Stemming from this objective are the following main contributions of this thesis:

1. A holistic approach to the design of flexible CMP systems on FPGAs is provided. Such an approach quantifies and captures important high level design options such that it is possible to formally model the design space. This is a basis for the development of automated methods which ultimately enables such systems to be efficiently designed.
2. Mathematical models for such systems are provided. These models provide a foundation for the

development of optimization techniques that are vital for designing competitive architectures through automation. Cost models makes it possible to reduce the complexity of the design process since tools can be employed to optimize cost functions, thereby reducing the work load that designers would otherwise need to spend using workbench-based exploration tools.

3. An evaluation of optimization methods for the synthesis problem is given. This helps to identify which techniques are feasible. In total, three methods are evaluated. The first studies general tractability using well established Integer Linear Programming. The second investigates the effectiveness of techniques from the domain of Boolean Satisfiability (SAT) for solving this particular synthesis problem. The third considers application-specific heuristics. In particular, this thesis shows under which conditions the synthesis problem can be optimally solved using greedy algorithms. Further this work explains the limitation of greedy algorithms for the synthesis problem, and three greedy-like heuristics are introduced. These heuristics attempt to achieve two things: first, to alleviate the limitation, and second, to provide a means for fast synthesis when greedy algorithms are not applicable at all.

The sum of these contributions constitute the solution for solving challenges arising from the second design phase outlined in figure 1.3.

1.4 Organization

The rest of this thesis is organized as follows: Chapter 2 introduces the design space of flexible CMP systems on FPGAs, followed by a discussion of corresponding design space exploration methodologies, where related work is put in context. The latter serves the purpose of identifying strengths and weaknesses of existing synthesis techniques that can be potentially employed for the problem at hand.

Chapter 3 proposes a design flow that addresses design objectives identified in this thesis. Furthermore, an Integer Linear Programming (ILP) model that captures important design aspects of flexible CMP systems is introduced. The model captures resource allocation, task-mapping, scheduling and makespan optimization, the emphasis being that these subproblems must be jointly solved to arrive at globally optimum solutions. An evaluation of synthesis via ILP is also given in this chapter, where limitations are shown.

Chapter 4 briefly introduces SAT techniques as well as a related field of Answer Set Programming (ASP). The chapter proceeds to discuss how the synthesis problem can be modeled as ASP-programs, and concludes by evaluating the effectiveness of SAT-based techniques as used in ASP for solving this synthesis problem. The merits of ASP over ILP are shown, and limitations are discussed.

Chapter 5 discusses in detail application-specific greedy-like heuristics. A formal proof is given using matroid theory to show that the synthesis problem can be optimally solved using greedy algorithms under certain conditions. Limitations of greedy algorithms are discussed, providing a motivation to explore problem resolution using heuristics which attempt to exploit characteristics of the synthesis problem. Moreover, these heuristics are evaluated, limitations are identified, and a synthesis strategy in the presence of those challenges is proposed.

Finally, concluding remarks are provided in chapter 6. The tool that has been developed in the course of this thesis is described in Appendix B.

2 Design Space of CMP Systems

This chapter introduces the design space of flexible CMP systems in order to identify parameters which need to be captured in a design flow. The introduction also serves as a background for a discussion on related work later in the chapter in order to single out strengths and weaknesses of existing design techniques that can be potentially used or adapted for designing such systems.

Figure 2.1 is a coarse presentation of the design space. Each arrow represents a dimension in the space which stands for design choices that can be made. Some design choices are not mutually exclusive: for instance, the on-chip network can be a hybrid of buses and point-to-point connections. A particular CMP system is a point in the space which, in the presentation, can be perceived by drawing a curve going through the points in each arrow as shown in the figure for a three-core architecture. The design space consists of three subspaces stemming from the corresponding major design activities : the system architecture, the programming paradigm, and the physical layout. All these subspaces have an impact on overall system performance. Moreover, the subspaces are not orthogonal so that design decisions made in one of them influence the effectiveness of those made in others. This is the reason as to why emphasis is placed on joint optimization of design parameters.

There is consequently a need to determine which dimensions must be included in explorations in order to avoid pre-constraining the design space since otherwise suboptimum architectures can result. This is the subject of the next three subsections. Methods for covering the design space as well as the discussion on related work follow the introduction of the design space.

2.1 System Architecture Subspace

This subspace offers options for high-level CMP system design. The dimensions reflect the choice of IP components to be included in the system.

2.1.1 Peripherals

Peripherals dictate how the CMP system will interact with its environment. In practical applications, a CMP system is by itself a subsystem so that peripherals are potential bottlenecks that impose a performance cap. There is typically a higher level partitioning of the application that is already in place. For instance in an SDR system, such a partition could map baseband processing to the CMP system, and higher communication layers to a single general purpose processor. In such a scenario, sufficient bandwidth must be guaranteed between the GPP, the flexible CMP, and the radio frequency frontend. Therefore, the choice of peripherals is a design decision that can be excluded from optimizations, under the assumption that peripherals that can meet system requirements exist and have already been optimally selected during high-level partitioning. This choice is an optimization parameter at system level that should be dealt with in that context.

2.1.2 Processing Elements

2.1.2.1 Type and Architecture Style

Processing elements determine the time that will be spent executing parallel tasks. In addition to the general architecture of the processing elements (general purpose, signal processing, application-

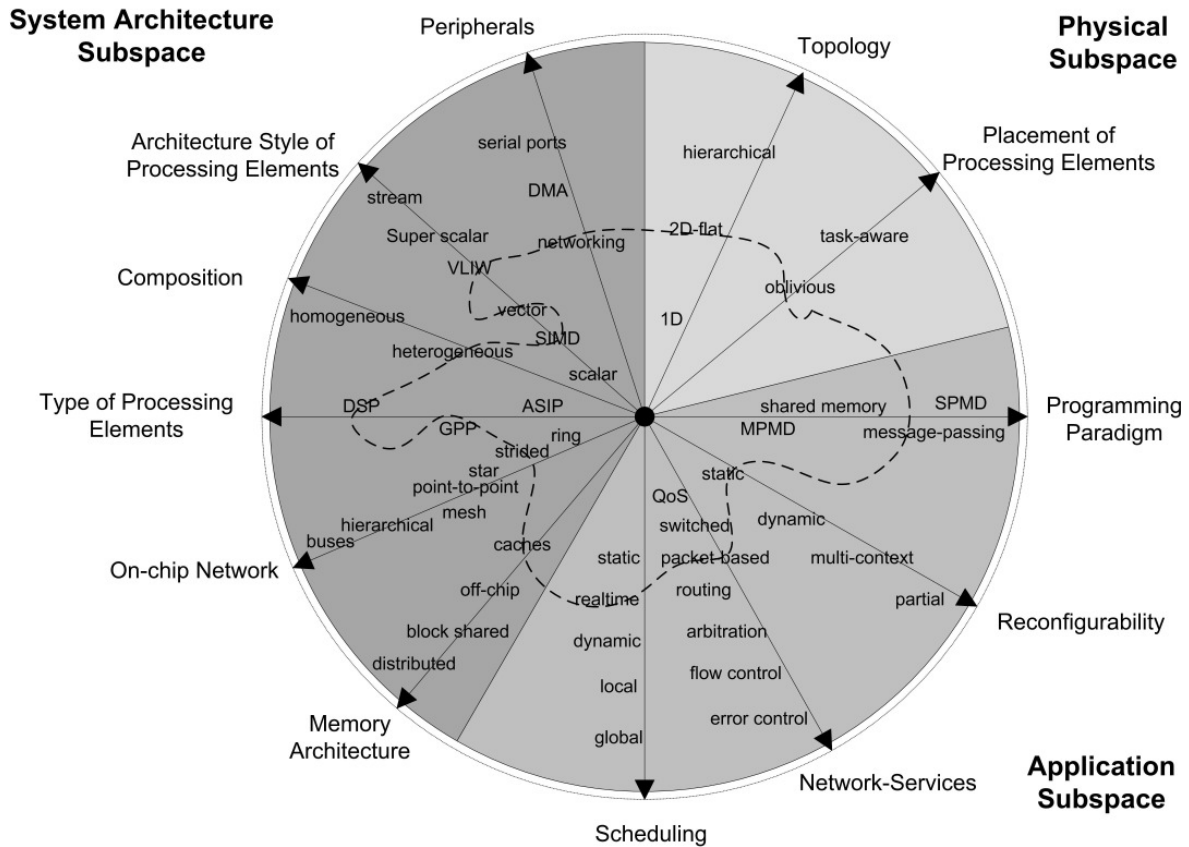


Figure 2.1: CMP design space

specific), there is a finer choice in the architecture style (scalar, SIMD, VLIW, vector, etc.). As depicted in figure 1.1, the tradeoff that needs to be made in the general architecture is between flexibility, execution performance, area and power consumption. However, for the exploration of flexible CMP systems, flexibility is not considered because processing elements can always be swapped to fit new applications.

The tradeoff in the style is similar involving performance, area and power consumption, but excluding flexibility. Scalar RISC processors are simple and thus are small in area, so that they are well suited for general purpose applications. SIMD processors are more suitable for exploiting data-level parallelism, but they tend to have large register files because of required parallel processing. VLIW processors can be used well when the application exhibit instruction-level parallelism, but they also tend to suffer from large register files due to the need for several parallel read/write ports. Super scalar processors do well where instruction-level parallelism is not visible at compile time, but that comes with a considerable control overhead in the micro-architecture which is manifested in higher area and power consumption. Vector processors are comparable to SIMDs, they have a much larger number of parallel processing units. They are well suitable for large scientific applications.

The question as to which processing elements are more suitable for a given parallel program largely depends on the kind of computation that is done within tasks. For the purpose of design space exploration, not all levels of architectural details must be captured, but the sum of their effects as visible in the actual performance and area cost. Parameters which need to be captured for each processing element are

Computation cost of each task in the parallel program. In addition to the clock cycles spent for actual computation, the cost include latencies due to memory accesses, and any speed-up due to loosely coupled hardware accelerators. Here, processing elements are composed of the pro-

cessor micro-architecture, local memories, interfaces and optionally accelerators as depicted in Figure 1.2. This inclusion is arbitrary, but is otherwise not limiting as long as the effects of these components are captured.

Cycles can be obtained via cycle accurate simulations, however that could be prohibitive for Multi-Program Multi-Data (MPMD) paradigm when the number of tasks and processors is large since the cost must be obtained for each possible processor-task pair. Other means of obtaining cycles include statistical estimation [16] and macro modeling [17]. Cycle estimation methods based on modeling can be used to lessen potentially costly simulations. Their downside is that they are applicable for well characterized processors only. Moreover, such characterizations may not capture processor features specifically designed for particular application domains.

Clock cycle lengths are challenging to estimate when soft cores are used, because the critical path also depends on placement and routing algorithms. Reliable figures can be obtained when FPGA placement constraints are in place. Otherwise it may be necessary to conduct an exploration iteratively in order to refine cost models. This technique is proposed in Chapter 3.

The area cost of the processors. This includes external resources processors need such as memories, accelerators and controllers. The area can be obtained through FPGA synthesis for the target platform. Again, placement constraints are required to get reliable figures, or alternately, iterative flows should be used.

The number of each available type of processors. This implies the use of an IP library. To avoid pre-constraining, the library should ideally be extensive.

With this parameter, the tradeoff is between the degree of task-level parallelism that can be exploited at the system level, and hence the performance, against the area. However, a higher degree of parallelism could degrade the overall performance due to the need for data transportation and synchronization between the processing elements. This crosscutting issue is an example for non-orthogonality of the design space.

Here, a distinction is made between the number of parallel tasks in the application, and the number of processors. The former parameter is a separate tradeoff, and is in the application subspace. The latter parameter influence the extent to which tasks can share a processor. This introduces yet another crosscutting issue with scheduling in the application subspace.

2.1.2.2 Composition

The choice between homogenous or heterogenous composition of processing elements directly impact the area-performance tradeoff. Processing elements in homogenous architectures tend to be more general purpose so that any task can be mapped onto them. On the other hand, heterogenous architectures include specialized processors, leading to more efficient execution of specific tasks. Consequently, the latter style tends to be more favorable from the performance point of view.

Furthermore, the composition trades performance against usability, but not always so. If the parallel tasks in the application perform different sequences of operations as in multiple program multiple data paradigm, then heterogenous architectures generally would lead to better performance because each task can be mapped to a processor that is best suited to it. On the other hand, if the parallel tasks as essentially the same performing the same sequence of operations on different data, then a heterogenous architecture may not offer any advantage, unless there is a limit of a number of cores of a particular type or style that can be used.

The downside is that programming for specialized processors is a difficult task that often requires non-generic tools. Nevertheless, this is a problem that will be solved by adequate tools so that usability need not be considered during exploration.

Targeting applications to heterogenous architectures is inherently more difficult since different tools need to be incorporated in the design flow to cater for different processor flavors. Despite the challenge, the IP library in use for exploration should be rich enough in order to efficiently execute tasks on optimum processors. The final composition must result from exploration, and not the other way round (i.e. make decision on composition, and then find a suitable exploration methodology).

No parameters need to be modeled for this dimension. A design flow must only insure the richness of the IP library, ideally through a preliminary analysis that uses designer's expertise and application knowledge to identify possible candidates.

2.1.3 Communication Infrastructure

On-chip communication infrastructure is a critical component of a CMP system as it impacts the application performance and power consumption. The reason is non-scalability of metal wires used for interconnection, leading to wire delays which are significantly longer than gate delays. The result is high dynamic power consumption due to large capacitances and also it becomes expensive to move data between processing elements, and to or from memories and peripherals. Thus, the system architecture must strike a balance between parallelization and data movement.

The tradeoff between network architectures is between average performance, scalability and area. Links are the simplest of all networks from protocol perspective. Direct links between nodes (i.e. processing elements, memories and peripherals) have the highest possible performance because the bandwidth is not shared. Moreover, transmission overhead is minimum as there is no need for control and management of the communication medium.

But since the number of links can grow exponentially with the number of nodes depending on the application, the area for direct links do not scale well. Another disadvantage is that the area requirement at the nodes also grows with the number of links because multiplexors must be used to direct traffic to different links. Worse still, multiplexors increase the capacitive load at the end of links so that buffers must be employed to maintain link performance. With crossbars, multiplexor overhead can be moved into the communication infrastructure. However, the fundamental area problem arising from exponential growth remains.

The best area efficiency is obtained with buses, but their performance does not scale well because nodes must share bandwidth. Control techniques such as pipelining and out-of-order completion of transactions mitigate the effects of bandwidth sharing, but do not solve the fundamental problem resulting from bandwidth sharing. Better results can be obtained with hierarchical buses since these exploit traffic locality to reduce contention on global buses. Bridges are then typically used to route traffic between hierarchies so that overall, area is traded for average performance.

A generalization of hierarchical buses leads to the concept of Network-on-Chip (NoC). These are more adaptable to applications since their topologies and protocols can be selected to suite a specific traffic pattern. However, their complex nature introduce cross-cutting issues: whereas throughput maybe improved, multiple hops increase latencies. Requirements for Quality of Service (QoS) introduce an additional dimension in the application subspace because routing algorithms must be involved. These in turn directly impact the the latency and throughput.

Throughout this thesis, the term network is used to refer to any kind of on-chip communication resource (link, bus, NoC, etc.). The term communication infrastructure refers to all networks that constitute a CMP system.

The selection of a suitable communication infrastructure depends not only on the traffic pattern characteristic of the parallel program, but also on selected processing elements. Therefore, these selections should not be conducted independent of each other because of the crosscutting issues between parallelization and data movements. Parameters which need to be captured for the communication infrastructure are

Type and number of networks. This implies the use of an IP library. There should be no restriction with respect to the number and type of networks that can be used in the system from the functionality point of view. This is important because the traffic pattern and the selection of processing elements may mandate the use of a hybrid network to get optimum results.

Model of communication cost between any two points in the network. This is required to get the cost in terms of time required to move data between any two tasks via that specific network expressed per unit volume of data. The dynamic nature arising from network protocols introduce additional challenges in estimating this parameter. These are discussed in Section 2.2.

Data traffic. This information is required to obtain a complete picture on communication cost. If communication cost depends on data volume transmitted, data traffic should be used to derive communication cost. Such variations occur when the ratio between control or communication overhead to payload is not constant. This aspect highlights another challenge due to dynamic nature as discussed in Section 2.2.

Area cost. This is required to determine overall CMP area consumption. The area should include the cost due to bridges, routers and adaptors. Similarly to processing elements, the area of a network is only an estimate that depends not only on FPGA resources, but also on placement and routing algorithms. Several iterations may be required to refine area estimates.

A unique area challenge for the communication infrastructure is that the parameter depends on the number of nodes. Therefore, the area should ideally be expressed as a function of number of nodes attached to it. Alternately, area costs can be modeled through different networks of the same type which support different number of nodes.

Capacity. Networks have a maximum number of nodes that can be attached to it. This poses a hard constraint during exploration that must be observed for each network.

2.1.4 Memory Architecture

Memories are traditionally a bottleneck because they are typically slower than processors hence leading to long access latencies, and because they have limited number of ports so that parallel accesses to them cannot be effectively exploited.

Memory hierarchies are used to trade performance for size where data or instruction locality can be exploited. Moreover, cache sizes and policies in a specific hierarchy can be used for performance tradeoffs: large caches can lead to a better performance since that can reduce cache misses, and higher associativity trade complexity, and thus area, for performance. Similarly, replacement policies and flushing mechanisms trade complexity for performance.

In addition to hierarchies, memory partitioning can boost performance by allowing more parallel accesses than would otherwise be possible with a physically contiguous memory particularly where data-level parallelism can be exploited, for example, in image and signal processing.

Finally, a choice between shared and distributed memories needs to be made. The former can boost the performance when large amounts of data must be passed to another processor, but sharing is a potential bottleneck. The latter eliminate bottlenecks due to sharing, but is associated with data movement overhead thus crosscutting with the communication infrastructure dimension. Another crosscutting issue is with programming paradigm in the application subspace: shared memories are naturally more suitable for a shared memory programming paradigm, whereas the latter is more suitable with a message passing paradigm.

For the purpose of flexible CMP exploration, modeling memory parameters is rather difficult because memory architectures tend to be transparent at higher programming application levels. One solution to this problem is to include the effect of memories to the cost of executing a task on a processor as

discussed in Section 2.1.2.1. However, that approach inherently assumes distributed memory architectures. To capture shared memories, one solution is to model memory nodes as processing nodes to which only storage tasks can be mapped to. Storage tasks can be perceived to be such tasks which do not process data, but simply serve as a common place to store shared data. Simulations must be conducted for such nodes to capture dynamic effects pertaining to access latencies. Alternately, statistical models can be employed.

2.2 Application Subspace

This subspace influence performance by determining how processing, storage and communication resources of a CMP system are utilized. Apart from crosscutting issues, the dimensions in this subspace do not represent tradeoffs on their own (i.e. area vs performance): only the performance is impacted¹.

2.2.1 Task Mapping

Task mapping is a lead design activity in this subspace, and conceptually precedes all other decisions in this subspace apart from programming paradigm. This is because optimizations for scheduling, data transfers and reconfigurability can only be made once it is known which task is supposed to be running on which processor.

The only parameter that needs to be captured is which task can be mapped to which processor. For example, storage tasks can only be mapped to memory nodes as discussed in Section 2.1.4. Other than that, the exploration methodology must allow arbitrary mappings. Prefixing mappings, for instance by utilizing application knowledge, may lead to suboptimum results when other dimensions are factored in. As an example, one cannot simply dictate that a correlation tasks must run on a DSP only because the DSP can execute the computation more efficiently compared to a GPP. Doing so favor local maxima but runs a risk of missing global maxima.

2.2.2 Scheduling

This dimension determine the utilization of processing elements, and crosscuts with the dimension for number of elements as introduced in Section 2.1.2.1. Scheduling is required when tasks must share a processor, either because there is a limited number of processors, because data movements are expensive, or because realtime requirements dictate so. Scheduling insures that shared resources are properly utilized, i.e., all tasks get a fair or a required amount of time slices, and that deadlocking does not result. Following distinctions can be made between different scheduling mechanism:

Cooperative vs preemptive. In cooperative schedules, each task decides when to let others use processors, for example, as in basic cyclic executives. This is a simplest of all scheduling mechanisms. Its disadvantages include (1) unfair use because a task can consume most time slices disregarding others, (2) blocking because a task may not release the processor, (3) difficult to maintain if a certain allocation of processor time among tasks is required because the allocation has to be manually programmed, and (4) important for performance, cooperative scheduling is wasteful because tasks can unnecessarily block a processor while waiting for data or other resources.

Preemptive schedulers on the other hand forcefully suspend task execution so that others can run. These schedulers are complex, but they eliminate the disadvantages outlined above.

¹There is a tradeoff between performance and energy dissipation, however this is not in the context of this thesis as discussed in Section 1.2

In complex applications, a preemptive scheduler is almost always the most suitable choice so that the decision does not need to go into an exploration cycle. Such schedulers inherently introduce an overhead because they consume CPU time. Depending on the scheduler, the overhead may vary depending on the number of tasks. For instance, a scheduler may need to search through or sort a list of pending tasks to determine which one has the highest priority. The time required for this operation naturally depends on the number of tasks, leading to a phenomenon known as scheduling jitter.

The overhead must be captured either analytically or through simulations because it impacts the performance. However, analytical determination is not possible where no assumptions on occurrence of events can be made.

Realtime vs non-realtime. In realtime systems, a scheduler must insure that task deadlines can be met to guarantee overall system functionality as is the case in many embedded systems. This aspect crosscuts with mapping by imposing restrictions on which group of tasks maybe mapped on the same processor.

Through mapping constraints, realtime scheduling can lead to worse performance by necessitating expensive data movements between tasks that must be mapped on different processors. Nevertheless, this is not necessarily a disadvantage because there are often no benefits in improvements beyond meeting realtime requirements.

Conversely, even though non-realtime scheduling can vastly improve the overall execution time by avoiding expensive data movements, that benefit could be of little use from the functionality point of view if task deadlines are violated.

This parameter must be captured as a constraint. Performance consequences result automatically and they therefore require no special considerations (i.e. these go into overall cost function for optimization).

Fixed vs dynamic priority. Preemptive schedulers often require task priorities to make decisions by letting higher priority tasks run first. This can improve performance if critical tasks are assigned higher priorities. Priorities can be fixed or dynamic depending on whether priorities can change at runtime².

Dynamic scheduling is complex and difficult to implement, but has the advantage that deadlines can be guaranteed at higher CPU utilization compared to fixed priority scheduling. This can increase the performance by avoiding data transfers over the communication infrastructure by allowing more tasks to be mapped on processors as would otherwise be possible with fixed priority scheduling.

This parameter must be captured either analytically or through simulations because it impacts the performance.

Mono vs multiprocessor scheduling. In mono processor scheduling, decisions on which task should run on which processor is done during the mapping phase. The scheduler merely operates on those decisions to derive optimum schedules for each processor based on which tasks should be running on them.

On the other hand, a task can be released to run on any of the available processors in multiprocessor scheduling. This technique is essentially a dynamic form of task mapping.

The design decision on which of these two scheduling mechanisms is optimum depends on the characteristic of the application. Multiprocessor scheduling is more suitable in applications which consist of many aperiodic tasks because no knowledge on the rate of occurrence of events

²Measures against priority inversion do change priorities dynamically, but these are not considered to be dynamic scheduling schemes

is known at design time as is the case in personal computing and in some control applications. In the other case, that information can be exploited at design time to derive optimum tasks and schedules.

This parameter does not need to go into the exploration cycle because application knowledge can be used to make the selection in advance.

Global vs local. Local scheduling optimizes the usage of a processor between a group of tasks disregarding schedules for other processors. Global scheduling on the other hand attempts to schedule across processor boundaries to obtain overall better performance. In cooperative multitasking, global scheduling can exploit knowledge of availability of data between tasks on different processors, and is therefore in a position to derive better schedules for individual processors. This advantage disappears however in preemptive scheduling because critical tasks can always be released on data availability based on their priorities.

Therefore, unless cooperative scheduling is in place, this parameter does not require consideration.

2.2.3 Network-Services

Protocol features of networks impact system performance because they inherently introduce overhead or latencies. Options for network services are usually tied to the available network so that the selection of a network implies the selection of associated services. As a result, there is no direct tradeoff, however, the overhead that is introduced by the services needs to be captured. This section summarizes how different options affect the overhead.

Arbitration. In most cases, nodes must contend for network access so that mechanism that guarantee exclusive accesses must be in place. Therefore, arbitration is a problem of resource allocation or contention resolution. Arbitration is typically present in simple networks such as buses.

In collision avoidance mechanisms, nodes attempt to detect when the network is free so that they can transmit. When not free, a retry is attempted after a random waiting period. The overhead has three components: the detection which is constant, the random wait period which is unknown but is bounded, and the number of retries which is non-deterministic.

Priority based mechanisms are different in that a node will not backup if it has the highest priority among the currently contending nodes. This improves the overhead for highest priority tasks by eliminating random wait and retry components.

Networks which use an arbiter has an entity that is responsible for granting accesses which may or may not be based on priorities. The arbiter eliminates detection and retry overhead, leaving the unknown but bounded waiting period³.

Switching. Network switching can also be perceived to be a problem of resource allocation or contention resolution [18], and is typically present in complex multi-hop networks. In either case, switching determines when and how data is forwarded to its destination. Figure 2.2 shows a categorization of switching techniques.

In circuit switching, data follows pre-reserved paths to the destination. In exclusive mode, data transfer proceeds in three steps (1) the path to the destination is setup during which flow control information is routed to the destination, (2) the actual data is transferred along the reserved route without routing and control overhead, and (3) the reserved path is teared down. The overhead is in the first and last steps. For both setup and tear down, the overhead is indeterministic because

³A bound must exist, or there must exist an acceptable probability about an arbitrarily selected bound. Otherwise, functionality cannot be guaranteed, and the architecture is seriously flawed.

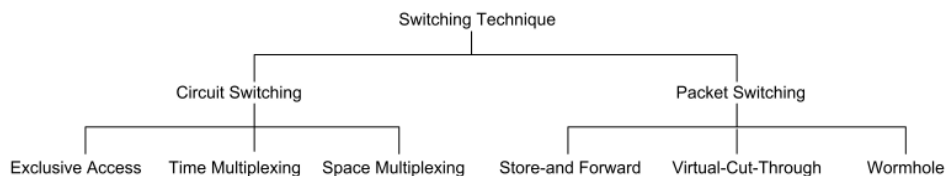


Figure 2.2: Network Switching Techniques

the routing path is unknown a priori. Additionally for setup, resources could be blocked leading to unknown waiting periods or retries.

In spatial multiplexing mode, there are virtual channels which are used to share the same physical path in order to increase resource utilization compared to the exclusive access mode. This is advantageous in situations where a node cannot fully utilize the bandwidth during the time when the physical path is reserved for it. A constant overhead in step (2) above is incurred because additional flow control is required to distinguish the virtual channels. Additionally, if care is not taken to insure that the bandwidth of reserved physical paths is not exceeded, additional delays will result in the step since contention may result.

In time multiplexing mode, physical paths are shared through the use of time slots. Therefore, in addition to the physical path, time slots are reserved as well in step (1). This eliminates potential overhead due to contention as is the case with spatial multiplexing.

In packet switching, data is split into packets which are independently routed to the destination, possibly along different paths. This further improves network utilization by eliminating reservation of resources, thereby eliminating the first and last steps above. The downside is that a stochastic overhead is introduced because of different arrival times of the individual packets caused by differing path lengths and contention of resources, and because of their different order of arrival which necessitates rearranging at the destination node. This is in addition to a constant overhead due to flow control information in packet headers.

The difference between the three packet switching techniques is in packet forwarding. In Store-And-Forward (SAF), a packet must completely arrive at a router before it can be forwarded to the next. This introduces a latency between routers which is non-deterministic because the number of hops for the packets varies randomly.

Virtual-Cut-Through (VCT) improves the latency by starting to forward a packet as soon as the header containing routing information has been received. Nevertheless, the latency remains indeterministic. The disadvantage of SAF and VCT is that routers require enough buffering space to store complete packets. This is also the case with VCT since a router must be able to store a complete packet in case the next router cannot yet accept forwarding.

Wormhole switching eliminates the requirement for maximum buffer size in routers by splitting a packet into smaller units. As in VCT, a packet is forwarded as soon as the header is received, however, forwarding is now done in smaller units, so that buffering space is required for the smaller units only. If a smaller unit cannot be forwarded, buffers can become filled up along the route back to the sending node, thereby stalling the entire stream of the packet. Since the blocking of a packet occurs simultaneously in several routers, deadlocking can result where several packets wait for each other to clear buffer spaces. Employing virtual channels efficiently solves deadlocking since packets can overtake blocked ones [19]. The associated overhead is the same as that in step (2) in spatial multiplexing.

Routing Algorithms. Routing is present in multi-hop networks to establish a path between sending and receiving nodes. Additionally, routing algorithms resolve contentions and attempt to achieve network load balancing to improve overall throughput. These algorithms fall into three categories.

Deterministic routing always provide the same path between nodes. However, overheads due to contentions remain non-deterministic. Oblivious routing selects a path randomly. Though this method is associated with all uncertainties in overhead cost, it has the advantage of achieving better load balancing compared to deterministic routing. Adaptive routing further improves on oblivious routing by considering local network state in path selection. From the perspective of deterministic overheads, nothing changes (i.e. the overhead is indeterministic).

Error Control. Transient faults have been shown to account for 80% of system failures in deep submicron VLSI circuits [20]. These can corrupt data along its transmission route so that link layer mechanisms are required to handle such errors.

With error detection and recovery, redundant information is added in the data stream to enable the receiving node to detect errors and request retransmissions. In this case, there is a constant overhead due to redundancy and detection, and a non-deterministic overhead due to retransmissions.

With forward error correction, enough redundant information is injected in the data stream so that the receiver can correct errors up to a certain number of wrong bits, in addition to the detection. The constant overhead is consequently larger, but that significantly reduces the non-deterministic overhead.

Quality of Service (QoS). Complex networks require a guarantee that transmission requirements can always be met in presence of errors and non-deterministic variations in overhead, latency and throughput. Thus in contrast to foregoing items, QoS does not result in overheads which must be considered in exploration cycles, but rather provide their bounds by way of guarantees so that an exploration cycle can operate on upper bounds instead. That way, resulting CMP systems are optimum for the worst rather than average or typical case. QoS is insured along four lines.

The first is data integrity and reliability by way of guaranteed bit error rates so that upper bounds on overheads resulting on transient faults can be computed based on the amount of data to be transferred.

The second is loss probability. This is related to error control in that dropped packets result in retransmission overheads. However, on-chip networks do not yet include packet dropping because of associated latency and throughput penalties.

The third is in-order delivery of packets through deterministic routing and flow control so that packets follow the same route without losses. This eliminates rearrangement overhead.

The last is network-wide resource management to guarantee upper bounds on latencies and throughput in presence of previously discussed non-deterministic factors.

2.2.3.1 Obtaining Network Parameters

The stochastic behavior of network services imply that the overhead can only be estimated for the average case if adequate analysis or simulations are conducted. Moreover, the overhead is not a fixed parameter for a network, but depends strongly on the communication pattern of the parallel program, especially with respect to the contention of network resources. This is because, in presence of scheduling and communications within the CMP environment, it is currently not possible to precisely determine the time at which *every* task may begin to transmit data.

If it is known that the estimated overhead is relatively small because of the large amount of data to be transmitted, the uncertainty can be expected to have marginal effects on the overall system performance so that overheads can be ignored in exploration cycles. If that is not the case, the designer has an option between worst-case and average-case optimization. The first option would base on QoS guarantees, whereas the latter on average case estimates.

Adequate simulations that reflect the communication pattern of the parallel application are very expensive for the exploration of flexible CMP systems because of the need to conduct such simulations for every mapping option of interest. The situation is particularly hopeless for multi-hop networks where the spatial position of nodes in the network matters because a completely different network behavior can result from the same communication pattern of the parallel program merely by changing the location of processors relative to each other. For N communicating tasks, and M possible nodes, the number of combinations that would need to be evaluated is

$$NM + (N-1)MN \frac{N!}{2!(N-2)!} + \dots + 2M \frac{N!}{(N-1)} = NM + M(N-1)^2 N + \dots + 2MN \rightarrow O(N^{2N}) \quad (2.1)$$

This exponential complexity renders such an evaluation computationally impossible for many practical cases. Worst-case parameters can be obtained for random traffic pattern under network steady state conditions by using the setup outlined in [19] which is depicted in Figure 2.3.

In the setup, packet sources send random data to destinations through the network. The monitor counts and time-stamps packets to determine network throughput and latency. Both parameters increase as the traffic into the network is increased from zero because of a better network utilization and more packet collisions respectively. There exist a point, the saturation point, beyond which the throughput no longer increases and the latency rises sharply. In some networks, the throughput will drop beyond the saturation point so that congestion control measures must be employed.

The saturation point gives worst-case parameters for QoS guarantees. Parameters obtained in this way will be almost always pessimistic taking into account application-specific communication pattern. Therefore, a designer may decide to use actual communication pattern of the parallel program instead of random traffic. Even though a full exploration is not possible, better network parameters may be obtained in this way. For instance, a designer may use a placement heuristic to place nodes with heavy traffic between them close together in the network.

To further reduce analysis time, finite state machines can be used as packet sources and destinations in the setup of Figure 2.3 rather than full task simulations as proposed in [21]. Simulation time is reduced by eliminating computations in tasks: the finite state machines emulate the behavior of tasks from the communication perspective.

Analytical methods based on network flow analysis can be used instead of simulations, however, such techniques are only suitable with oblivious and deterministic routing [18] so that they are generally not applicable for the exploration of flexible CMP systems.

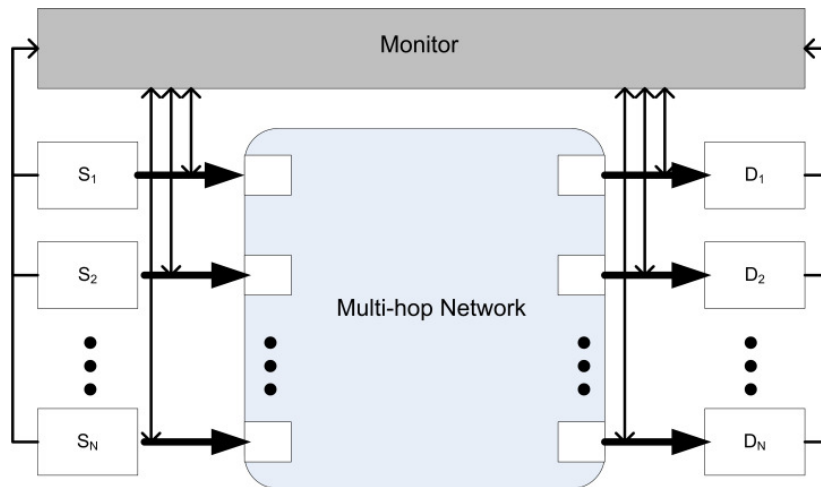


Figure 2.3: Setup for extraction of network parameters

2.2.4 Reconfigurability

Reconfiguration enables temporal sharing of FPGA resources so that there is a time overhead when switching between configurations. Applications which are affected are those which require dynamic reconfiguration during their execution, for instance, to adapt the architecture to prevailing conditions. The overhead is constant for any given FPGA architecture if the whole FPGA is reconfigured, and can be obtained from data sheets. The magnitude depends on the reconfiguration scheme used, where multi-context FPGAs tend to be much faster to reconfigure than their single-context counter parts.

On the other hand, the overhead for partial reconfiguration naturally depends on the FPGA area to be replaced. However, with current technologies, FPGAs cannot be arbitrarily reconfigured, but rather, an area for partial reconfiguration must be allocated. All components that will be dynamically reconfigured must fit in the pre-allocated space. Consequently, the overhead is constant for a given application. An estimate of the area consumed for dynamically reconfigurable components must be obtained prior to exploration to determine the overhead.

2.2.5 Programming Paradigm

The programming paradigm dictates how CMP resources will be used, and thus has a role of pre-selecting CMP architecture styles.

Shared memory model. In this model, parallel tasks exchange data via a shared memory. Typically, there is a single OS image for all processors that manages system resources. This models naturally fits with shared memory architectures which were discussed in Section 2.1.4. The on-chip network is rather simple because communications go through shared memories.

Message passing model. In this model, data is physically moved from one address space to another. Typically, each processor has its own OS image, which may be the same. This model naturally fits with distributed memories, and often requires elaborate network services in presence of many cores to manage inter-processor communications.

Single Program Multiple Data Streams. In this model, each processor executes the same program which perform the same sequence of operations on different data streams, for instance, as in embarrassingly parallel applications. Since there is little inter-processor communications in such scenarios, this model fits naturally with distributed memories and with rather simple on-chip networks.

Multiple Programs Single Data Stream. In this model, the processors executes different sequence of operations on the same data stream, for instance, as in signal processing. This model is applicable together with both shared memory and message passing models.

Thus, designer's knowledge on the application at hand can be used to prune the design space prior to exploration by selecting suitable components for on-chip networks and memories that will go into the IP library.

2.3 Physical Subspace

The physical subspace determines how components are placed on the chip, and is thus conceptually a design activity that is done in lower levels, particularly in placement and routing. However, there are crosscutting issues with system level design that affect the effectiveness of decisions made in either level. The network topology ultimately determine the optimum layout that placement and routing algorithms will find. Conversely, results obtained after this step may invalidate assumptions on network

parameters made in system level. Moreover, these algorithms do not have any information concerning the application. The layout may thus be optimum with respect to achievable clock frequencies, but knowing the amount of traffic between nodes as well as the work load of processors is vital to achieving better results. This information can be indirectly communicated to the algorithms through design constraints. In either case, it may be necessary to feed information obtained from placement and routing back to high level exploration.

2.4 Design Space Exploration

After reviewing the design space for flexible CMP system, this section discusses methods for exploring the design space. The discussion on related work is categorized in this chapter based on both the parameters covered and the exploration methodology used.

Design Space Exploration (DSE) consists of three distinct design steps, which are the specification of the design functionality, the architecture specification, and architecture implementation and design evaluation. The functionality is specified independent of the other steps. Architecture specification is derived from the functionality, i.e. the architecture is specified such that the intended functionality is covered. In the implementation step, the architecture specification is translated to obtain either a formal description of the architecture, or the actual physical implementation. The last step is accompanied by evaluations to determine whether the implementation meets design objectives with respect to the functionality and constraints. Evaluation results are also used to compare architecture alternatives in terms of design objectives for the purpose of optimization. The advantage of the separation of the overall design activity into these three steps is that designers can focus their activities to one of the steps independently of the other two.

As shown in Figure 2.4, a systematic DSE process operates on three distinct data-sets: one or several descriptions of intended functionality (application), description of feasible architectures and evaluation results from performance analysis. The three above-mentioned design steps operate on these data-sets. A detailed descriptions of each of these steps can be found in [1].

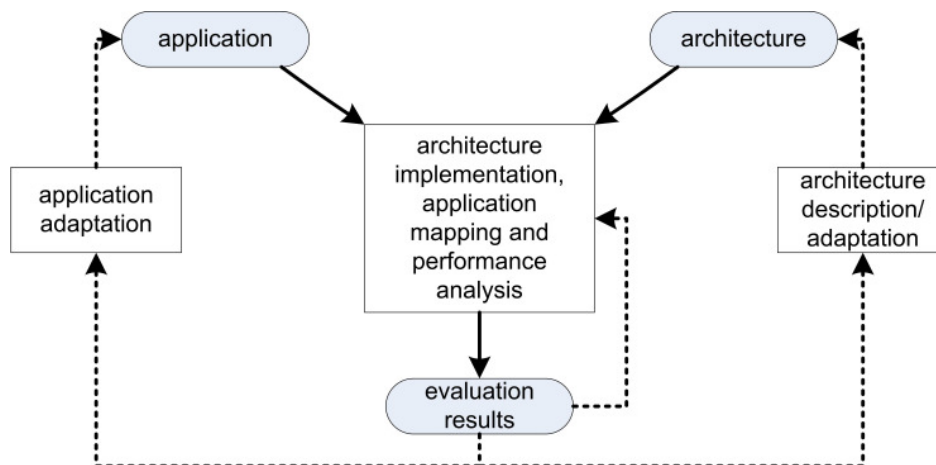


Figure 2.4: Y-Shaped design space exploration

The purpose of DSE is to expose efficient design points by optimizing design objectives. This is primarily a search and compare procedure following the Y-shaped flow in Figure 2.4: mapping options and design parameters are systematically altered, and evaluation results are ranked or charted against each other to find superior designs. The notion of design efficiency in pareto sense is used when multiple conflicting design objectives need to be traded against each other [1, 22, 23]. Superiority is defined according to the Pareto criterion for dominance [24] which states that a design is superior to

another if it is better in at least one objective, while being at least the same in all other objectives. Consequently, a design is optimum (or efficient) if it is not dominated by any other design [24]. As depicted in Figure 2.5, exploration methods can be categorized according to three hierarchies.

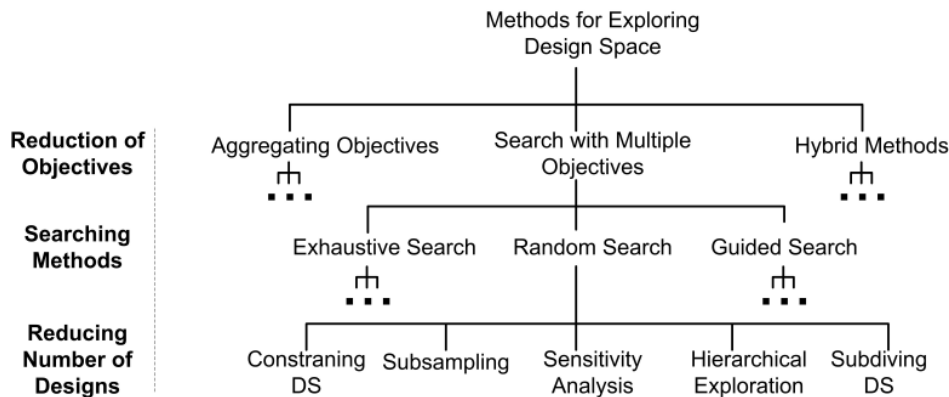


Figure 2.5: Methods for exploring the design space

Reduction of objectives. A central aspect of DSE is the optimization of objectives. Principally, there exist two methods for optimizations with multiple objectives. In the first, objectives are simultaneously optimized using multi-objective optimization techniques. Because such techniques are difficult to handle in many practical cases, the second method attempts to aggregate several objectives into one so that simpler single-objective optimization techniques can be employed.

The categorization in the first hierarchy is about whether a DSE method uses aggregation. Whereas the reduction of objective functions through aggregation enables the use of simpler single-objective optimization methods, they may not reveal all efficient design points (the so-called non-supported efficient solutions [25]).

Search methods. Since DSE is a search procedure, the second hierarchy categorizes DSE methods depending on how they conduct the search. For each method of reducing objectives, exhaustive, random or guided search can be used as shown in Figure 2.5.

With the first search method, all possible design points are described, implemented and evaluated. Because the resulting effort may be prohibitive, a subset of all feasible designs can be randomly selected for implementation and evaluation using the second method. Random selection can be further subdivided into blind and metric-driven methods. Blind methods select parameters disregarding any previous findings or exploration state. Metric-driven methods on the other hand use criteria in the selection process. For instance, simulated annealing uses a probability function that depend on the energy of two states to make a transition from one state to the other.

The last method differs from the former two in that knowledge from evaluation of one design point is used to decide how design parameters should be modified to get efficient solutions as depicted in Figure 2.6(a).

Reducing number of designs. Since search procedures are often expensive in terms of computation time, DSE methods usually attempt to reduce the time by evaluating a smaller number of designs. The categorization in this hierarchy is about how the number of designs are reduced during exploration. With each of the search method shown in figure 2.5, the number of designs for implementation and evaluation can be reduced through any of the following five techniques.

Constraining the design space means that the number of design parameters or their range of values is restricted to a region of interest. This technique utilizes designer's knowledge on the

application. Sub-sampling reduces the number of designs by quantizing the value of parameters in a desired range. Sensitivity analysis and hierarchical exploration techniques attempt to prioritize certain design parameters so that those parameters with less influence on design objectives can remain unexplored. The last technique reduces the complexity by dividing the problem into subproblems which are then independently explored. In this way, not all combinations of parameters need to be explored. This method is illustrated in figure 2.6(b).

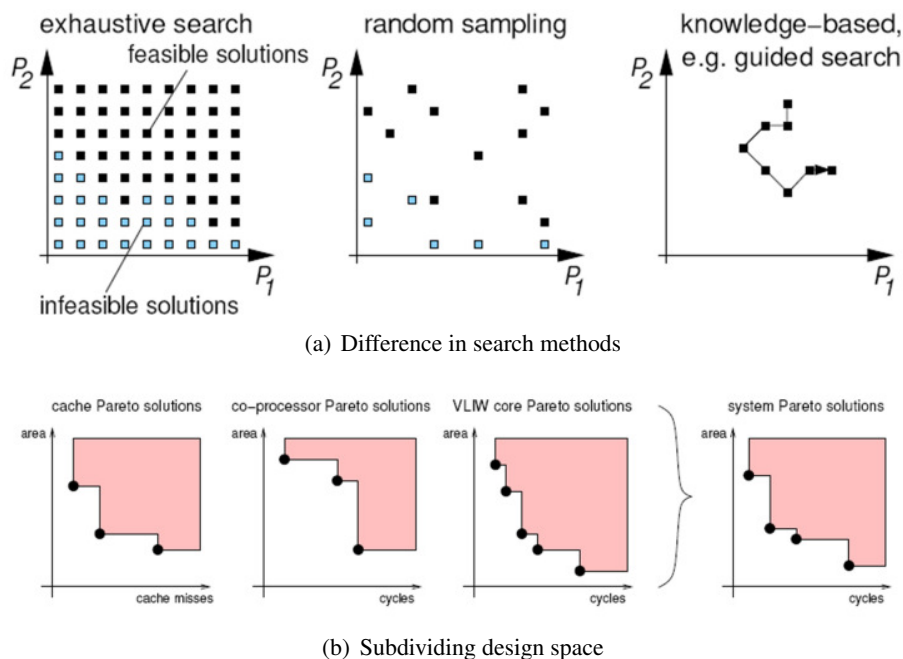


Figure 2.6: Illustration of exploration methods (source[1])

2.5 Related Work

Published literature on various aspects pertaining to the design of CMP systems based on design space exploration is vast. This section attempts to categorize related work in this area in order to identify strengths and weaknesses of various approaches that can be potentially adapted to designing flexible CMP systems.

2.5.1 Processor Design and Exploration

Work in this category aim at developing IP cores for use in CMP (or SoC) systems. In addition to exploration work leading to soft cores such as ARM and MIPS processors, several publications describe corresponding design methodologies such as [26, 27, 28, 29, 30]. These design methodologies use guided search and/or employ techniques for reducing the number of designs. The use of such methodologies in the scope of multiprocessor design is limited to the subproblem of designing constituent IP cores.

In some cases, processor exploration frameworks are integrated into CMP design tools to serve as instruction set simulators [31, 32] so that these tools can be used for explorations based on guided search. In particular, tasks can be mapped to simulators followed by simulations to evaluate the mapping. However, this methodology is impractical for our purpose given the huge size of the flexible

CMP system design space. In another context, such simulators can be used for functional verifications of flexible CMP systems as in the proposed design flow as described in Section 3.1. Generic frameworks for system analysis such as [33] can also be employed for this purpose.

2.5.2 NoC Design and Exploration

Work in this category aims at developing efficient NoC architectures as well as understanding their behavior for CMP systems. Development work can be subdivided into work that propose architectures and corresponding techniques for network services such as [34, 35, 36], those that describe design methodologies such as [37, 38, 39], as well as simulative and analytical evaluation methodologies as in [40, 41, 42]. A comprehensive survey on NoC is reported in [43].

Similarly to processor design tools, the applicability of these NoC design tools and methodologies is limited to the IP design subproblem. Their potential use during the search of the CMP space is rendered impractical by the size of the design space. However, tools that can simulate NoC behavior such as [44, 45] are important for functional verifications of the final CMP architecture.

2.5.3 CMP Design Tools and Methodologies

Work in this category aim at addressing CMP design challenges by proposing design methodologies and corresponding tool-support, typically targeting the system architecture subspace. Platform-based design is a central methodology that is commonly used.

Platform-based design is an emerging paradigm that aims at accelerating implementation. The paradigm follows the Y-shaped exploration methodology (Figure 2.4). Architecture components are described in an IP library such that there exist one or several descriptions for each component. These descriptions correspond to different abstraction layers of the platforms or of its components. Thus, mapping and evaluation is conducted starting at high abstraction layers with subsequent refinements into lower layers until a fully implemented optimum design is obtained. The exploration itself is either manual using guided search, or is automated using random search.

Methodologies which build on manual guided search [46, 47, 48, 31, 49] have the inherent disadvantage of being impractical where the design space is huge as is the case with flexible CMP, therefore these are not of interest.

Other authors have developed methodologies that build on random search to handle the complexity problem. These approaches rely on optimization of a cost function using combinatorial optimizations, simulated annealing, evolutionary algorithms or application-specific heuristics. While these tools take the step towards the right direction by eliminating tedious manual explorations, they are limited for flexible CMP design because they do not consider cross-effects between the subspaces as they only target specific dimensions. As a result, suboptimum results are likely. The following examples, while not exhaustive, demonstrate this general weakness that is inherent in published design methodologies:

Net-Chip [50] uses a heuristic algorithm to map cores on communication topologies such as mesh, torus etc. The objective function can be the minimization of network hop-delay, area or power consumption under bandwidth and area constraints. However, the mapping of tasks on cores is not considered.

UMARs [38] is a single objective algorithm for mapping, routing and slot allocation in NoC-based architectures. The algorithm attempts to map cores to NoC architectures such that latency and bandwidth are met without over allocating network resources. The approach has the same limitation as Net-Chip in that application mapping to cores is fixed.

MOCDEX [51] is a multi-objective design space exploration methodology that attempts to integrate

system design and placement and routing. The approach generates a random CMP configuration, and proceeds with synthesis, results of which are used for ranking. This is a very interesting approach because it directly include actual synthesis results to determine Pareto optimum architectures. The drawback is that system design is limited to the allocation of hardware resources to components: for instance, allocating cache sizes of individual processors out of FPGA's block RAM. The system architecture is otherwise fixed.

Magellan [52] framework applies a machine-learning approach for the optimization problem. The machine-learning algorithm derives its intelligence by simulating benchmarks on available processor cores. This is an attractive technique because exploration time can be significantly reduced after initial simulations. However, the framework does not consider communication effects.

Kumar et al. [53] uses dynamic programming to synthesize an architecture, taking into consideration task mapping and the allocation of communication resources. Moreover, context switching overhead is considered so that this is one of the most complete automated approach for CMP design. However, the approach is limiting because it assumes shared memory communications.

Jin et al. [54] used integer linear programming for makespan optimizations. Task mapping is considered to simultaneously allocate processors and map tasks onto them. However, they only assume point-to-point communications.

Beltrame et al. [55] uses a unique approach based on automated guided search. Rather than using simulations for every considered mapping, the approach attempts to estimate the impact of varying architectural parameters probabilistically. Simulations are conducted only when the estimates are insufficient. Consequently, the number of expensive simulations is reduced. However, this approach is only applicable where an estimation methodology exists. Obviously, that is possible with architecture templates, but not with arbitrary CMP architectures.

Meyer et al. [56] use simulated annealing to balance performance and cost by optimally distributing and sharing memories and communication resources. Memory allocation and bus topology exploration are jointly conducted making decisions on where the data should be located. The approach is limited by fixed task mapping.

Dick [57] presents in his dissertation four algorithms for automatic synthesis targeting distributed embedded systems, platform based design, embedded client-server architectures and reconfigurable systems respectively. The first two algorithms are also one of the most complete approaches as they tackle allocation, mapping and scheduling problems. The last two are not related to CMP synthesis as they target communications over wireless links and co-processing on FPGAs respectively.

The first algorithm is a genetic multi-objective algorithm. Even though processing elements are treated as discrete integrated circuits, that is not a limitation for use with CMP systems. To adapt the algorithm for CMP synthesis, one needs to extend it to cover arbitrary NoC because the algorithm assumes a specific fixed communication infrastructure. The second algorithm, while specifically targets CMP systems by including the provisioning of clock signals and the generation of layout, does not lift the limitation imposed therein for communication infrastructure.

Zhu et al. [58], Lee et al. [59] and Ha et al. [60] present approaches that tackle the allocation of processors and communication resources, together with task mapping. Principally, arbitrary network resources can be used in presented algorithms. Their drawback is that the three sub-problems are disjointedly solved because of ranking: e.g. processor allocation first, mapping second, and finally allocation of communication infrastructure. This approach does not capture cross-effects between the dimensions so that sub-optimum results are likely.

2.5.4 Mapping Applications to Fixed CMP

The focus in this category is not to derive or synthesize CMP architectures, but rather to explore ways of mapping applications on existing platforms or templates [61, 62, 11]. Since only the application subspace is targeted, results obtained are only local optima in the subspace. In addition to manual guided search [63], mapping algorithms such as evolutionary [64] and simulated annealing [65] have been used. Other approaches have used subdivision of the design space [66] to reduce mapping complexity.

Scheduling during or after mapping has been extensively treated in the literature [67, 68]. There also have been efforts to map tasks in a way that optimizes for power [69], reduces chip temperature at run time [70], or minimizing inter-processor communications [71]. Dynamic mapping techniques has also been introduced with objectives such as temperature management [72] and performance optimizations through adaptive mapping [73].

2.5.5 Domain-Specific Exploration

For completion, we mention the last category in the CMP design field whose intent is to determine efficient SoCs for a given domain or set of algorithms [74, 75, 76]. The result of these efforts are platforms that can be targeted by one of the methods discussed in Section 2.5.4.

2.6 Chapter Summary

Three conclusions can be made from the discussion in this chapter:

Automation is necessary. Automated methods must be used to cope with design complexity. Whereas a skilled engineer can effectively utilize work-bench based tools to design a feasible architecture, the sheer number of design parameters renders a disciplined exploration infeasible. As demonstrated later in Chapter 3, often no consistent trend with respect to design objectives can be observed when design parameters are systematically changed, where the objective can increase or decrease by more than two orders of magnitude when moving from one parameter set to another. Moreover, results obtained can be counter intuitive. Consequently, even experienced designers cannot effectively execute a guided exploration based on their expertise because it is not easy to predict the outcome of parameter variation. Those results underline the need for an automated approach.

Inclusive modeling of the design space is necessary. Methodologies for solving individual sub-problems have been well studied. Ideas from these can be adapted for flexible CMP synthesis. However, care must be taken in modeling the problem to insure that the design space is not constrained. Leaving out some parameters from the model ignores important cross-effects between dimensions, and between subspaces.

A distinguishing aspect of architecture synthesis as proposed in this work is a framework which enables an automated design space exploration based on mathematical models for flexible CMP systems. This allows a wider and a disciplined coverage of the design space by relieving designers from tedious manual coverage, and hence avoids constraining the design space.

Joint coverage of subspaces is necessary. Ranking individual dimensions or subspaces by optimizing one of them after the other has the same effect of constraining the design space as excluding design space dimensions because cross-effects are ignored. Therefore, when determining the system architecture according to Figure 1.2, it is important to simultaneously (i) select processors (ii) map and schedule tasks to them, and (iii) select one or several networks for communications, such that design constraints and objectives are met.

Moreover, judicial exploitation of application knowledge and designer expertise should be used to prune the design space of unnecessary dimensions. The following chapter discusses how the design space can be mathematically modeled to cover important and necessary dimensions and parameters as identified in this chapter.

3 Problem Formulation

Chapter 2 introduced the design space of flexible CMP systems where parameters that need to be captured in a design flow were identified. This chapter discusses mathematical models for the parameters with the intent of enabling automated synthesis in order to fulfil the research objective mentioned in Section 1.2. Since the synthesis problem at hand is inherently combinatorial, an Integer Linear Programming (ILP) formulation is used for modeling. The evaluation of the feasibility of ILP-based synthesis is also given in this chapter. The same ILP model is a basis for other synthesis methods presented in Chapters 4 and 5.

Designing a flexible CMP system which is optimum for a specific application is an activity that inherently assumes that the rate of occurrence of events in the parallel program is known as is the case in signal and image processing applications, as well as in mathematical and scientific computing. Applications with unpredictable behavior are better served with dynamic optimization methods. This thesis focuses on the former class of parallel programs. We begin by proposing a synthesis flow based on ILP in Section 3.1. The model itself is presented in three parts in Section 3.2.

3.1 Proposed Design Flow for Flexible CMP Systems

Figure 3.1 illustrates the proposed design flow for flexible CMP systems that covers application mapping, architecture determination, system integration, and FPGA synthesis.

The input to the flow is a parallel program and optionally information on task periods and deadlines for realtime requirements. The program is executed and analyzed to obtain inter-task data traffic and task precedence information. The execution is purely functional and can be conducted on any processor without altering the required information¹. Similarly to other related work in this area, the other input to the design flow is an IP library containing information on available processing elements and communication networks, as well as their costs and constraints. This information is used to specify an instance of an ILP problem which is subsequently solved by existing solvers.

The solution to the ILP instance is then used to generate an abstract description of the system which is passed to PinHat to generate the configuration bit-stream. The description passed to PinHat specifies the number and type of processors, task mapping and their schedules on processors, allocated networks and information on which task uses which communication network for a specific inter-task data transfer.

Because post-synthesis results could deviate from initial cost models used as discussed in Section 2.3, new cost models can optionally be extracted after placement and routing to start a new iteration. The generated description can be optionally passed to a simulation framework for the verification of the functionality.

For real time systems, it is often sufficient to meet timing constraints so that the interest is not to find the fastest solution. In such situations, the ILP formulation in this flow can be used to find the smallest system instead by reversing the roles of the performance objective and that of the area constraint.

¹The information required is the data traffic pattern, which is a function of the parallel program only.

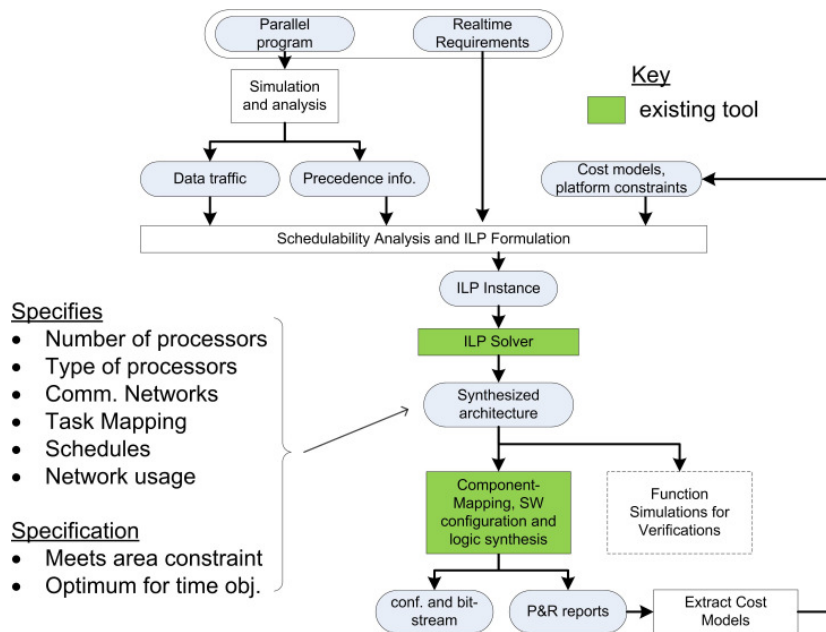


Figure 3.1: Proposed architecture synthesis flow

Design space parameters from Chapter 2 (Figure 2.1) are captured as follows in this flow:

- Peripherals are not captured in the ILP model because that information is not available in the application. Communication ports and off-chip memories are specified in PinHat as lower level information.
- Architecture style and type of processing elements are captured in the IP library.
- Composition results explicitly from ILP solution.
- Network-types are captured in the IP library.
- Memory architecture is captured through processing elements in the IP library as depicted in Figure 1.2. As discussed in Section 2.1.4, shared memories require dedicated processing elements which can be included in the IP library.
- Scheduling constraints are captured through realtime requirements. Their costs are computed from task-mappings and from processor parameters in the IP library as discussed later in section 3.2.2.
- Network services are implicitly specified in the IP library. Their cost is computed from the communication traffic and from network parameters as discussed later in Section 3.2.1.4.
- Reconfigurability is not captured because of limitations in prevailing technologies. Nevertheless, this does not constrain the design space because the overhead can be trivially added to the model once reconfigurability issues have been solved.
- Programming paradigm is a parameter that is not automated in this flow. Designer's knowledge on the application is used to specify suitable components in the IP-library that best suit the used programming paradigm as discussed in Section 2.2.5
- Physical subspace is captured through the feedback by way of post-synthesis extraction of cost models as discussed in Section 2.3.

Capturing these parameters in the flow enables a non-constraining and a joint optimization methodology as is desired to meet the objective set in earlier chapters.

3.2 ILP Formulation

Next we describe an ILP model for automated synthesis for flexible CMP systems targeted at architecture determination. The optimization problem is to:

1. Map parallel tasks to a set of existing processing elements (processors) and in the process to select the number and type of processors
2. To generate a suitable communication architecture, which can be a mixture of different network topologies
3. Schedule the tasks on processors, when needed, to meet realtime requirements

such that a given FPGA area constraint is met, and the execution performance of the parallel program is optimized, where the optimality criteria is as defined in 1.2. The ILP formulation is described in three parts: Section 3.2.1 describes the basic model for points 1 and 2 above, Section 3.2.2 extends the basic model to include scheduling, and finally, Section 3.2.3 describes a generalization of the performance objective for overall computation time of a program, or for makespan minimization. Each part is concluded by experimental results aimed at studying the feasibility of ILP-based synthesis. Results obtained are the basis of discussions on alternate synthesis methods in Chapters 4 and 5.

The following notation is used: $\mathcal{T} = \{I_0, \dots, I_n\}$ is a set of tasks in the parallel program. $I_i \in \mathcal{T}$ is the i th task in the set. Similarly, $J_j \in \{J_0, \dots, J_m\}$ is a processor with the index j in an IP library, and $C_k \in \{C_0, \dots, C_K\}$ is a communication resource in the library with an index k . The binary decision variable $x_{ij} = \{0, 1\}$ represent a task mapping option such that $x_{ij} = 1$ means task I_i is mapped on processor J_j , 0 otherwise.

3.2.1 Basic Formulation

The basic formulation covers constraints that insures functionality, without any regard to deadlines or critical paths.

3.2.1.1 Task-Mapping Constraint

The following constraint specifies that all tasks must be mapped, and that each task must be mapped exactly once:

$$\sum_{j=0}^m x_{ij} = 1, \forall I_i \quad (3.1)$$

With the above constraint, a task I_i is mapped on a specific type of a processor J_j . But we have not specified how many instantiations of a processor J_j we have. The number of instances is a parameter that corresponds to the composition dimension in Figure 2.1 on page 8. If we were to introduce a decision variable which specifies how many instances of a processor we can have, then we would additionally need to formulate a constraint that specifies that for each task-processor mapping, exactly one instance must be used. Otherwise, (3.1) becomes ambiguous since it is not clear on which instance a task should be mapped to. The corresponding constraint is given by

$$x_{ij} \leq \sum_{b=0}^{B_j} instance_{jb}, \forall I_i, J_j \quad (3.2)$$

where $instance_{jb}$ is the decision variable for a b th instance of processor J_j , and B_j is the corresponding number of instances. However, such a formulation, while necessary, would lead to a non-closed formulation because the summation bound K_j is itself a decision variable which needs to be determined. In contrast, a combinatorial problem must be bounded in the sense that the solution space is known a priori.

To get to a closed formulation, we allow multiple instances of the same processor in the processor set $\mathcal{J} = \{J_0, \dots, J_m\}$ so that each instance of the processor will have its own index j . Therefore, when we formulate the problem we can arbitrarily fix the number of instances K_j of a processor type. Nevertheless, this is not a limitation in practise, because we can, in principle, select K_j to be arbitrarily large prior to solving the optimization problem.

3.2.1.2 Processor Sharing Constraint and Cost

It is often the case that several tasks need to share a processor, for example, because of area constraints, or to minimize communication costs. We thus must specify how many tasks can be mapped onto the same processor instance J_j , where the limit is imposed by the available instruction and data memory. Here we make the following assumptions which are valid for systems under consideration:

1. There is a task scheduler such that the optimization problem does not involve scheduling. This restriction is lifted in Section 3.2.2.3.
2. The mapping of tasks is static, i.e. there is no run-time dependent dynamic mapping. This assumption is made because the rate of occurrence of events is known a priori so that dynamic mapping is not advantageous for the performance objective.
3. The sequence of instruction for each task is stored on a local processor memory at system initialization, i.e. there is no post-loading of instructions in a dynamic manner for tasks from, say, a shared system memory. This assumption does not exclude the use of instruction cache and address space extensions to off-chip memories. Its sole purpose is to facilitate deterministic calculations of memory requirements.
4. The number of tasks which can be mapped onto a processor is limited by the size of the corresponding instruction memory only, including any extended address space.
5. The cost of task switching in terms of computation time on a given processor is constant. This simplification is lifted in Section 3.2.2.3.

Consequently, we need a constraint for address space utilization and costs for task switching. Let s_{ij} be the size of task I_i in the program memory of processor J_j , and s_j be the size of the program memory. The address space constraint is formulated as

$$\sum_{i=0}^n x_{ij} \cdot s_{ij} \leq s_j, \forall J_j \quad (3.3)$$

meaning that the sum of address space requirements for all tasks mapped on a processor J_j does not exceed the memory capacity.

A distinction is made between cooperative multitasking and preemptive scheduling when modeling processor sharing cost. Let t_j be the cost of task switching on processor J_j . This parameter captures the time that is required to save the processor state associated with a currently running task, and to restore a previously saved state associated with another task. The total overhead is thus a multiple of t_j , depending on the number of times task switching takes place.

It is difficult with cooperative multitasking to determine that number without simulations because switching depends on the behavior of tasks, which in turn, generally depends on where the task is running. Consequently, each task must be simulated on each processor to obtain the parameter T_{ij} , which is the number of times the task will relinquish the processor resource. Therefore, assuming that a task will release a processor independent of what other tasks might be doing², the overhead is given by

$$T_{switch_{cooperative}} = \sum_{j=0}^m \left(t_j \cdot \sum_{i=0}^n T_{ij} \cdot x_{ij} \right) \quad (3.4)$$

In many practical cases however cooperative multitasking is not used because of its severe disadvantages. Widely used preemptive schedulers rely on a clock interrupt handler which is activated for scheduling activities at regular known intervals, T_{clock_j} . Consequently, the number of times a task will involuntarily relinquish processor resources is given by T_{ij}/T_{clock_j} , where T_{ij} is the duration of the task on the processor. That number is accurate because T_{ij} is the net duration which excludes task pending and suspension time when other tasks are running. The total cost of task switching is thus

$$T_{switch} = \sum_{j=0}^m \left(\frac{t_j}{T_{clock_j}} \cdot \sum_{i=0}^n T_{ij} \cdot x_{ij} \right) \quad (3.5)$$

3.2.1.3 Processor Area Constraint and Cost

Let a_j be the area of a processor J_j . Ignoring the communication network for a moment, the area constraint for an FPGA could be specified as

$$\sum_{j=0}^m \sum_{i=0}^n x_{ij} \cdot a_j \leq A_{PE} \quad (3.6)$$

where A_{PE} is the maximum area on FPGA for mapping processing elements. A drawback of this formulation is that if several tasks will share the processor, then the area for the processor will be counted several times, leading to a bias against processor sharing. To clarify this point, consider a case where we have two tasks and two instances of the same processor. The area cost of mapping the two tasks would be the same for both possible mappings so that the problem reduces to minimizing the computation time only, and hence disregarding time-area trade-off. Even though we do not explore time-area trade-off directly, this circumstance could render a problem infeasible because it might appear that the FPGA area is too small.

To compensate for resource sharing, we could try to weight the area to get the constraint

$$\sum_{j=0}^m \sum_{i=0}^n x_{ij} \cdot a_j \cdot \frac{s_{ij}}{s_j} \leq A_{PE} \quad (3.7)$$

so that virtually, a task uses a fraction of the area of a processor, based on the size of the task in the program memory of that processor. However, this work-around leads to the correct computation of area cost only when all processors are fully utilized. For partial utilizations, i.e. when more tasks can be mapped on the processor, the area cost of the processor would be incorrect.

A correct solution involves the introduction of a decision variable for a virtual processor v_j for specifying the area cost. For each instance J_j of a processor, we have a corresponding virtual processor v_j . Then, we specify that for all tasks that are to be mapped to a certain instance J_j , one, and only one, of the tasks must also be mapped to the corresponding virtual processor v_j . We then count the area

²The assumption is that a task release a processor based on its functionality and availability of data or other resources on the processor only.

of virtual processors instead. Note that such a counting cannot be done directly with real processors because every task must be mapped according to (3.1).

For mapping tasks on virtual processors, we cannot simply introduce a decision variable x_{iv_j} and specify a mapping constraint similar to (3.1) because a task can be mapped to a virtual processor which doesn't correspond to the real one. Therefore, a relationship between x_{ij} and v_j must be explicit as in the following logical constraint

$$x_{0j} \vee x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj} = v_j \quad \forall J_j \quad (3.8)$$

which essentially states that v_j must be 1 when at least one task is mapped on its corresponding real processor. Note that what we really want is having one task only mapped on the virtual processor, however, an XOR relationship cannot be used because that would introduce mutual exclusiveness with task mapping on the real processor which is equivalent to constraining the design space.

Transforming the logical constraint into a linear equivalent is rather difficult. So we use instead a logical implication, whose linear transformation is discussed by Guret et al [77]. The constraint is thus

$$x_{0j} \vee x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj} \rightarrow v_j \quad \forall J_j \quad (3.9)$$

and its linear transformation is

$$\frac{1}{n+1} \sum_{i=0}^n x_{ij} \leq v_j \quad \forall J_j \quad (3.10)$$

Note that because of the scaling, the Left-Hand Side (LHS) of (3.10) is always less or equal to one. If at least one task is mapped on J_j , then the LHS is greater than zero, forcing v_j to equal one. However, if the LHS is zero, both $v_j = 0$ and $v_j = 1$ will lead to a feasible solution. This is a direct consequence of using a logical implication rather than the equality (3.9). But we want to force v_j to zero in such cases, otherwise we will have incorrect area. Therefore, we additionally add the constraint

$$v_j \leq \sum_{i=0}^n x_{ij} \quad \forall J_j \quad (3.11)$$

so that the area constraint is then exactly given by

$$\sum_{j=0}^m v_j \cdot a_j \leq A_{PE} \quad (3.12)$$

The overall area constraint is considered after formulating communication network constraints and costs.

3.2.1.4 Handling Interconnect

Next we formulate constraints and costs for the communication network. Also in this case the combinatorial problem consists of selecting specific type of networks for inter-task communications such as buses, rings, meshes, fat trees, hypercubes etc., under given constraints, but we allow different topologies to be mixed to avoid constraining the design space. The formulation challenge in this case stems from three aspects:

- Application-dependent dynamic communication patterns requires us to capture not only the traffic between two tasks, but also the frequency of transmissions in order estimate the overhead stemming from network services.
- Allowing the mixing of different networks in order to inclusively explore the design space.

- Allowing the arbitrary sharing of networks, meaning that two processors can be connected by more than one network, and that two sets of communicating tasks on two different processor may transmit their data over different networks. This prevents constraining the design space.

While complicating, these three aspects do not impose restrictions to network usage so that an optimum architecture is not excluded from the solution set of the optimization problem. Therefore, we assume for the following formulations that:

1. If two communicating tasks are mapped on the same instance of processor then they communicate locally only for example via a local shared memory, or through OS-dependent mechanisms (for example in-processor message passing using techniques such as queues). Further, we assume that intra-processor communications:
 - i.) Have negligible communication latency and overhead (can be easily relaxed however in presence of significant memory access latencies, for example, with off-chip memories).
 - ii.) Do not contribute to area cost on the FPGA since the area cost of memories is already accounted for by the parameter a_j in (3.12).
2. A processor has network interfaces to support available networks in the library, when necessary via bridges or adaptors as exemplified in Figure 3.2. However, it is not mandatory that a processor supports all of the available networks.
3. An instance of a processor can use more than one communication network, the limit being the physical network interface of the processor. However, it is not mandatory that the processor must use all of the supported networks.
4. A communication network has arbitration cost resulting from possible contentions of the resource by parallel tasks. Here, we assume that:
 - i.) Tasks which share a processor are not prioritized with respect to network access. In particular, this assumes a presence of queues at network interfaces. This simplifies the estimation of the overhead cost because that becomes independent of task pairings. Data packets can however have priorities at the application level for QoS guarantees within the network during routing. Moreover, data packets can have priorities at the application level that can be used for access arbitration purposes. The advantage is that overhead can be estimated without having to consider possible task pairings, which could lead to an explosion of decision variables.
 - ii.) We can always compute an upper bound on arbitration time for each network topology depending on the number of processor instances on it, or at least, we have an acceptable probability about an arbitrarily selected bound. Otherwise, the implementation has a fundamental functional flaw. This assumption also relies on QoS guarantees as discussed in Section 2.2.3.1.
 - iii.) We cannot predict when two or more tasks on different processors will attempt to access a network simultaneously, but we can assume a certain probability distribution depending on the structure of the graph representing the parallel application and its communication pattern.

Assumptions in item 4 above result from the non-deterministic³ nature of the system, and are not unique to automated synthesis approach as proposed in this thesis. Optimizing under the probability distribution and the arbitration bound gives a designer a mechanism to optimize for the average or for the worst-case.

³The non-deterministic nature with respect to data communication does not contradict the assumption made in the first paragraph in this chapter about deterministic application behavior with respect to the rate of occurrence of events.

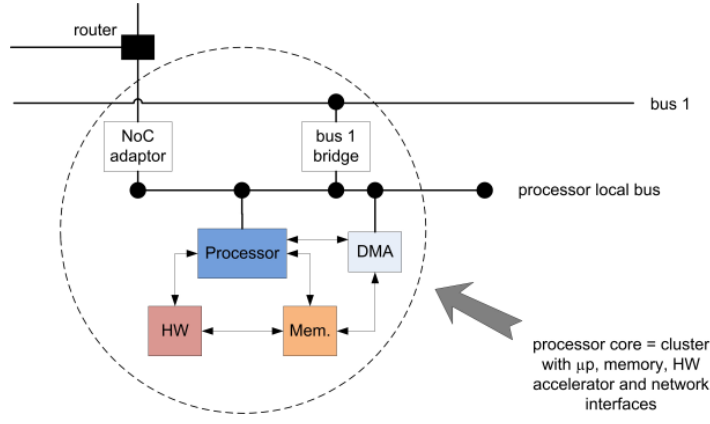


Figure 3.2: Processor core showing inclusion of network interfaces

3.2.1.5 Network Usage Constraints for Inter-Task Communications

With the assumptions in the previous subsection we are in a position to formulate constraints which insure that mapped tasks use communication networks. Let y_k be a binary decision variable with value 1 if a topology C_k shall be used for communication between any two tasks I_{i_1} and I_{i_2} , and 0 otherwise.

Now we first want to specify that exactly one communication topology must be selected for two communicating tasks mapped on different processor instances. There are two possibilities for any given pair of communicating tasks: both tasks are mapped on the same processor, or on different. For the latter case, there are $m \times (m + 1)$ different ways of mapping. Therefore, the following logical constraint results:

$$(x_{i_1 j_0} \wedge x_{i_2 j_1}) \vee (x_{i_1 j_0} \wedge x_{i_2 j_2}) \vee \dots \vee (x_{i_1 j_m} \wedge x_{i_2 j_{m-2}}) \vee (x_{i_1 j_m} \wedge x_{i_2 j_{m-1}}) = y_0 \oplus \dots \oplus y_K$$

$$\forall I_{i_1}, I_{i_2} \in \{I_0, \dots, I_n\} \mid I_{i_1} \prec I_{i_2} \text{ and } J_{j_1}, J_{j_2} \in \{J_0, \dots, J_m\} \mid J_{j_1} \neq J_{j_2} \quad (3.13)$$

Here, $I_{i_1} \prec I_{i_2}$ is a precedence operator meaning that I_{i_1} precedes by I_{i_2} in the directed application graph implying data transfer from task I_{i_1} to task I_{i_2} (modeled as a directed edge). The exclusive-or operator in the right-hand side specifies the condition that one and only one network must be selected in the case of mapping on different processors. For reasons which will be explained later, we first assume that we only have two communicating tasks. This restriction is lifted shortly. Because of (3.1), at most one of the conjunctive terms in the LHS of (3.13) is true, and therefore, the disjunction of the terms suffices.

Note that (3.13) is a composition of several sub-constraints. It is thus convenient to introduce 0-1 auxiliary decision variables to convert (3.13) into a series of linear constraints. Let $\alpha_{i_1 i_2 j_1 j_2}$ be such a variable, so that the following sub-constraint results

$$\alpha_{i_1 i_2 j_1 j_2} = x_{i_1 j_1} \wedge x_{i_2 j_2} \quad (3.14)$$

for a corresponding mapping, meaning that the variable is 1 only when the two communicating tasks are mapped on different processors. The linear transformation of the logical constraint according to the technique described by Guret et al. [77] yields two inequalities

$$\alpha_{i_1 i_2 j_1 j_2} \geq x_{i_1 j_1} + x_{i_2 j_2} - 1 \quad (3.15)$$

$$\alpha_{i_1 i_2 j_1 j_2} \leq \frac{x_{i_1 j_1} + x_{i_2 j_2}}{2} \quad (3.16)$$

Substituting for $\alpha_{i_1 i_2 j_1 j_2}$ in (3.13), we obtain

$$\alpha_{i_1 i_2 j_0 j_1} \vee \alpha_{i_1 i_2 j_0 j_2} \vee \dots \vee \alpha_{i_1 i_2 j_m j_{m-2}} \vee \alpha_{i_1 i_2 j_m j_{m-1}} = y_0 \oplus y_1 \oplus \dots \oplus y_K \quad (3.17)$$

Note that $\alpha_{i_1 i_2 j_1 j_2} \neq \alpha_{i_1 i_2 j_2 j_1}$. This distinction is required for these reasons:

1. Memory or scheduling constraints may allow $x_{i_{j_1}}$, but not $x_{i_{j_2}}$.
2. It may be cheaper to map I_i on J_{j_1} rather than J_{j_2} from a computation, communication or scheduling perspective.

Moreover, if the application graph is cyclic, then $\alpha_{i_1 i_2 j_1 j_2} \neq \alpha_{i_2 i_1 j_1 j_2}$, otherwise $\alpha_{i_2 i_1 j_1 j_2}$ does not exist in the ILP model. For cyclic application graphs, processor indices and order must be the same for both $\alpha_{i_1 i_2 j_* j_*}$ and $\alpha_{i_2 i_1 j_* j_*}$, else constraint (3.1) cannot be fulfilled.

Next we introduce an auxiliary 0-1 decision variable $\lambda_{i_1 i_2}$ for the LHS of (3.17). As mentioned above, only one of the $\alpha_{i_1 i_2 j_* j_*}$ can be true. At formulation time, it is not known which exactly. Therefore, $\lambda_{i_1 i_2}$ is a linking variable which is used later to formulate a logical constraint with reduced variables, whose linear transformation is easier to handle. A linear form for this linking variable which satisfies the LHS of (3.17) is simply

$$\lambda_{i_1 i_2} = \alpha_{i_1 i_2 j_0 j_1} + \alpha_{i_1 i_2 j_0 j_2} + \cdots + \alpha_{i_1 i_2 j_m j_{m-2}} + \alpha_{i_1 i_2 j_m j_{m-1}} \quad (3.18)$$

so that the following simpler logical constraint results from (3.17)

$$\lambda_{i_1 i_2} = y_0 \oplus y_1 \oplus \cdots \oplus y_K \quad (3.19)$$

The linking variable equals 1 only when the two communicating tasks are mapped on different processors. Note that if there are more than one pair of communicating tasks, then the formulation has the following undesirable property that if at least two communicating tasks are mapped on the same processor, then either:

- i All tasks must be mapped on the same processor, and hence enforcing sharing. This is because, in such a case, every y_k must equal zero; i.e. if a $\lambda_{i_1 i_2} = 0$, then no network can be allocated in order to fulfil the RHS of (3.19). Or,
- ii All networks must be used, because every y_k must equal one to fulfill the RHS.

To lift the previously made restriction on the number of communicating tasks, and to eliminate the above mentioned property in the formulation, we need to specify that either of the following two conditions must be fulfilled (but not necessarily both); if $\lambda_{i_1 i_2} = 0$, then we don't have a bound on a summation over y_k , i.e., y_k can be arbitrarily set to accommodate other communicating tasks, completely ignoring the current pair

$$\lambda_{i_1 i_2} = 0 \rightarrow \sum_{k=0}^K y_k \leq M' \quad (3.20)$$

or if $\lambda_{i_1 i_2} = 1$, then at least one y_k must be 1 to insure that a resource will be allocated for the pair

$$\lambda_{i_1 i_2} = 1 \rightarrow \sum_{k=0}^K y_k \geq 1 \quad (3.21)$$

where M' is a ‘‘very large’’ number. However, we have not yet stated which resource will be used, i.e., when more than one of the linking variable $\lambda_{i_1 i_2}$ equals one, and more than one resource is allocated, we have no way of telling which pair should use which resource. Therefore, an additional binary decision variable $z_{k i_1 i_2}$ is introduced. Its value is 1, if I_{i_1} and $I_{i_2} \mid I_{i_1} < I_{i_2}$ will communicate over

a topology C_k . With the variable, a set of constraints that insures that exactly one topology is used without the above restriction is thus

$$\sum_{k=0}^K z_{ki_1i_2} = \lambda_{i_1i_2} \forall I_{i_1}, I_{i_2} \mid I_{i_1} < I_{i_2} \quad (3.22)$$

$$y_k \geq z_{ki_1i_2} \forall C_k \text{ and } I_{i_1}, I_{i_2} \mid I_{i_1} < I_{i_2} \quad (3.23)$$

The first constraint forces one of the variables $z_{ki_1i_2}$ to take the value 1 when the communicating tasks are mapped on different processors, effectively making the allocation choice according to (3.21), while granting freedom according to (3.20). However, we still need the variable y_k to formulate area constraint to allow us to count the area only once, because more than one of the variables $z_{ki_1i_2}$ can equal one so that these cannot be used for counting the area. Therefore, the second constraint forces $y_k = 1$ when at least one of the $z_{ki_1i_2}$ equals 1.

Since a processor can support several but not necessarily all of the available networks, we specify that

$$\begin{aligned} \alpha_{i_1i_2j_1j_2} + z_{ki_1i_2} &\leq V_{j_1j_2k} \forall C_k \text{ and } I_{i_1}, I_{i_2} \mid I_{i_1} < I_{i_2}, J_{j_1}, J_{j_2} \in \{J_0, \dots, J_m\} \mid J_{j_1} \neq J_{j_2} \\ V_{j_1j_2k} &= \begin{cases} 2 & \text{if both } J_{j_1} \text{ and } J_{j_2} \text{ support } y_k \\ 1 & \text{otherwise} \end{cases} \end{aligned} \quad (3.24)$$

where $V_{j_1j_2k}$ is a parameter which insures that a selected network is supported by both processors to which the communicating task pair is mapped to. If $\alpha_{i_1i_2j_1j_2}$ is 1, then so is $\lambda_{i_1i_2}$ because of (3.18), from which follows one $z_{ki_1i_2}$ must equal one because of (3.22). If $V_{j_1j_2k}$ is 1, then either $\alpha_{i_1i_2j_1j_2}$ or $z_{ki_1i_2}$ or both must equal zero. If $\alpha_{i_1i_2j_1j_2} = 0$, then $z_{ki_1i_2}$ can be freely selected for a different mapping through another $\alpha_{i_1i_2j_*j_*}$, in which case $\lambda_{i_1i_2}$ still equals one. If $z_{ki_1i_2} = 0$, then a different network can be allocated through another $z_{ki_1i_2}$, provided there is an $\alpha_{i_1i_2j_*j_*}$ that equals one. Therefore, $V_{j_1j_2k} = 1$ rules out invalid network allocations without constraining the mapping. If on the other hand $V_{j_1j_2k} = 2$, then both $\alpha_{i_1i_2j_1j_2}$ and $z_{ki_1i_2}$ can equal one, which implies the network allocation for the communicating pair.

3.2.1.6 Network Capacity Constraint and Area Cost

Since some communication topologies have a maximum capacity with respect to the number of processors which can be attached to it, we additionally formulate the following constraint:

$$y_k + \sum_{I_{i_1}, I_{i_2} \mid I_{i_1} < I_{i_2}} z_{ki_1i_2} \leq M_k \forall C_k \quad (3.25)$$

where M_k is the maximum number of processors which can use a topology C_k . If A_{net} is the area constraint for routing resources on FPGA, and A_k is the area cost of a topology C_k , and A is the total FPGA area, then we additionally have

$$\sum_{k=0}^K A_k y_k \leq A_{net} \quad (3.26)$$

$$A_{PE} + A_{net} \leq A \quad (3.27)$$

3.2.1.7 Network Communication Cost

Next we need to formulate the cost of the topology in terms of computation time. For a given topology and a pair of communicating tasks, the cost should consider the inherent latency of the topology, and the amount of data transfer between the tasks. If we select a topology C_k for a pair of communicating

tasks I_{i_1} and $I_{i_2} \mid I_{i_1} < I_{i_2}$, then we also must pay for the communication cost over that topology. The variable $z_{ki_1i_2}$ captures the decision to pay for that cost.

The cost should also include the latencies through network bridges or adaptors which are attached to processors as exemplified in Figure 3.2 if these are significant. Since there are generally more than one such bridges or adaptors, an auxiliary binary decision variable is required to distinguish which pair of adaptors or bridges is being used at both ends of the processor. If $\beta_{ki_1i_2j_1j_2}$ is the variables, then the following set of constraints are required to properly set its value:

$$\beta_{ki_1i_2j_1j_2} \geq \alpha_{i_1i_2j_1j_2} + z_{ki_1i_2} - 1 \quad (3.28)$$

$$\beta_{ki_1i_2j_1j_2} \leq \frac{\alpha_{i_1i_2j_1j_2} + z_{ki_1i_2}}{2} \quad (3.29)$$

$$\forall C_k \text{ and, } I_{i_1}, I_{i_2} \mid I_{i_1} < I_{i_2}, J_{j_1}, J_{j_2} \in \{J_0, \dots, J_m\} \mid J_{j_1} \neq J_{j_2}$$

The total cost of communication between all tasks for the duration of the program (or for an iteration) T_{net} is thus given by

$$T_{net} = \sum_{I_{i_1}, I_{i_2} \mid I_{i_1} < I_{i_2}} \left(\sum_{k=0}^K \left((L_k D_{i_1i_2} + \tau_k p_k B_{i_1, i_2}) \cdot z_{ki_1i_2} + \sum_{j_1, j_2 \mid j_1 \neq j_2} \mathcal{L}_{kj_1j_2} D_{i_1i_2} \beta_{ki_1i_2j_1j_2} \right) \right) \quad (3.30)$$

Here, $D_{i_1i_2}$ is the total amount of data transferred between the two communicating tasks as obtained from functional simulations of the parallel program, and L_k is the transfer latency for network C_k for a word. Therefore, the first term in the innermost bracket captures the net communication cost. The overhead due to arbitration (including routing effects) is captured by the second term. Therein, p_k is the probability that network arbitration will be incurred when a task wants to communicate, τ_k is the bound on arbitration time as discussed in Section 3.2.1.4, and B_{i_1, i_2} is the number of transfers of data blocks between the corresponding tasks. Note that the overhead generally increases with the number of communicating tasks over a network as captured by $z_{ki_1i_2}$. The sum over processor indices captures the latency through network bridges or adaptors, where $\mathcal{L}_{kj_1j_2}$ is the sum of the latency of the two ends.

3.2.1.8 Objective Function

Initially, the objective is to minimize the overall computation time of the parallel program, and therefore we assume that programs either terminate or its tasks are periodic. Formulation for makespan optimization is described in Section 3.2.3.

$$\min \left(\sum_{i=0}^n \sum_{j=0}^m x_{ij} \cdot T_{ij} + T_{net} + T_{switch} \right) \quad (3.31)$$

3.2.1.9 Problem Size

Tables 3.1 and 3.2 summarize the problem size with respect to the number of constraints and variables respectively. Here, $N = n + 1$, $M = m + 1$ and $K' = K + 1$ are the number of tasks, processors and communication networks respectively. Therefore, a directed graph modeling a parallel application has N nodes. Let E be the number of edges. A non-linear increase in problem size is caused by formulating the cost for communication network ($z_{ki_1i_2}$ in (3.22) and (3.23)), and by arbitrary task mapping (x_{ij} in (3.1)), and $\beta_{ki_1i_2j_1j_2}$ in (3.28) and (3.29). Therefore, it is unlikely that problems with hundreds of processors and hundreds of different networks will be practically solvable. While such problems are currently unlikely in the embedded domain, it is desirable to address this problem for future systems which could be huge.

Table 3.1: Number of constraints

| From inequality | Number |
|---|------------------------|
| (3.1),(3.3),(3.10),(3.11),(3.12), (3.15),(3.18),(3.22), (3.25),(3.26),(3.27) | $N + 3M + K' + 3E + 3$ |
| (3.23),(3.24) | $2K'E + MN$ |
| (3.28),(3.28) | $2K'EM$ |

Table 3.2: Number of decision variables

| From inequality | Number |
|-----------------------------|---------------|
| (3.1) | MN |
| (3.10),(3.15),(3.18),(3.23) | $M + K' + 2E$ |
| (3.22) | $K'E$ |
| (3.28),(3.28) | $K'EM$ |

3.2.1.10 Experimental Results

This section presents experimental results aimed at studying the feasibility of ILP-based synthesis by analyzing both the synthesis time and quality of results produced. With the preceding ILP formulation, suitable applications for the experiment are those which do not have finishing deadlines, and which exhibit the same communication pattern for the same input data. This is because up to now we are considering non-realtime applications whose tasks are statically mapped. No requirement is otherwise made with respect to the structure of the parallel programs.

Six parallel programs which implement algorithms for common mathematical and physical problems were selected. The implementation style chosen is message passing using the Message Passing Interface (MPI) standard [78]. The choice is arbitrary: the formulation supports all styles. Table 3.3 summarizes the applications. The last column in the table describes the communication topologies from the perspective of tasks. The topology were arbitrarily selected for the programs. A selected topology determines the data traffic pattern and the work load in each task, and is thus a determining factor for the structure of the constraint matrix in the ILP instance, and consequently the run time.

Following MPI convention, each of the applications has a “master” task that distributes the computation between itself and “slave” tasks, depending on the number of parallel tasks and the topology. Figure 3.3 depicts the topologies. In the star topology, the master task is at the center, and distributes the load equally among the slaves. If a load remains after equal distribution among the slaves, it is processed by the master. The farming topology is similar to the star, but the load is not evenly distributed among the slaves: a slave is assigned a new load as soon as it is done processing its previously

Table 3.3: MPI applications in the experiment

| Application | Description | Topology |
|-------------|---|---------------|
| FIR | 100,000-point FIR-filter kernel | star |
| Derivation | Numerical derivation of functions using finite element method | farming |
| Simpons | Numerical integration using Simpons' method | star |
| Jacobi | Jacobi transformation of a 12x12 matrix | point2point |
| N-body | 2D N-body problem with 4000 particles | centered ring |
| Inversion | 2048x2048 matrix inversion | star |

assigned load. The master performs no processing beyond load distribution. The centered ring topology has the master at the center, and the slaves arranged in a ring around the master. The master sends data to one of the slave in the ring. The data is then iteratively processed, and is passed from one slave to the next in the ring until processing is completed. The result is sent by the last slave in the ring back to the master. These three implementations are scalable in the sense that a single program can be used for parallelization with different numbers of tasks. In the point2point topology selected for Jacobi, each task sends data to other tasks for further processing in an iterative manner, and is thus not scalable in the MPI sense since the number of tasks must be known beforehand.

Functional simulations were conducted to determine the data traffic between tasks. Functional simulations suffice because the data traffic does not vary with task mapping. Therefore, the MPICH2 [79] implementation of the MPI standard was used to simulate on a PC. The implementation is distributed with a library called MPE which can be used to obtain profiling data [79] as outlined in Figure 3.4(a). An MPE utility is used to convert the binary .clog2 output file containing profiling information into a textual format which is subsequently parsed to construct the application graph. Both the data traffic and task precedence information can be extracted from the file, therefore, the file is the input to the ILP formulation block which is depicted in Figure 3.1.

Figure 3.4(b) shows three types of entries in the file which contain the information used to construct an application graph. The first line contains information on a created task, in this case, with MPI rank 0. Therefore during construction, a task is created whenever such a line is found. A consequence of this is that all tasks can be captured even if some tasks do not communicate with others (i.e., when the application graph is not connected). Example of applications which fall under that category are trivially parallel Multiple Program Multiple Data. The second and third entries contain information on send and receive events. The MPI rank in the lines are associated to the sending and receiving ranks respectively. The size is given in bytes. Therefore during construction, the file is scanned to match send and receiving entries by comparing tag and size entries. Communicating tasks are thus identified by rank fields in matching lines. If multiple matching lines are found, then the edge between the nodes representing the tasks is weighted. The weight is the parameter $D_{i_1 i_2}$, and the number of matches is the parameter $B_{i_1 i_2}$ in T_{net} (3.30).

Task durations on processors T_{ij} were approximated by comparing MIPS parameters of processors through the following simple extrapolation

$$T_{ij} \approx \frac{\text{MIPS}_{J_j}}{\text{MIPS}_{T5500}} \cdot \text{Cycles}_{T5500} \quad (3.32)$$

where cycles on an Intel T5500 processor were obtained by direct execution on a PC. Ideally, cycle accurate simulations should be used to obtain this parameter. However, while inaccurate, this approximation does not affect much the general trend obtained for synthesis time.

The size s_{ij} of tasks on processors were similarly approximated by comparing word sizes of memories and extrapolating. A precise approach is to compile a task for each processor type. This estimation does not impact constraints (3.10) and (3.11) because the size of memories of the processors in the experiment were large enough to fit all tasks. However, using more accurate figures will change the run time because the constraint matrix will change. Nevertheless, the general trend observed is expected to be the same (i.e. the trend with respect to the solver runtime, not with respect to the solutions obtained).

Parameters on available processors and networks are read into the formulator from text files representing the IP library. Two types of general purpose RISC processor cores were used in this experiment: Xilinx Microblaze (MB) and PowerPC 405 (PPC). Because the PowerPC is a hard micro on the target Xilinx Virtex IV-FX FPGA, only two instances are usable. For networks, Xilinx's Fast Simplex Link (FSL) and On-chip Peripheral Bus (OPB) were used. Tables 3.4 and 3.5 show the parameters of processors and networks for the target FPGA.

The task switch time per scheduling interval t_j/T_{clock_j} is derived from the number of registers which

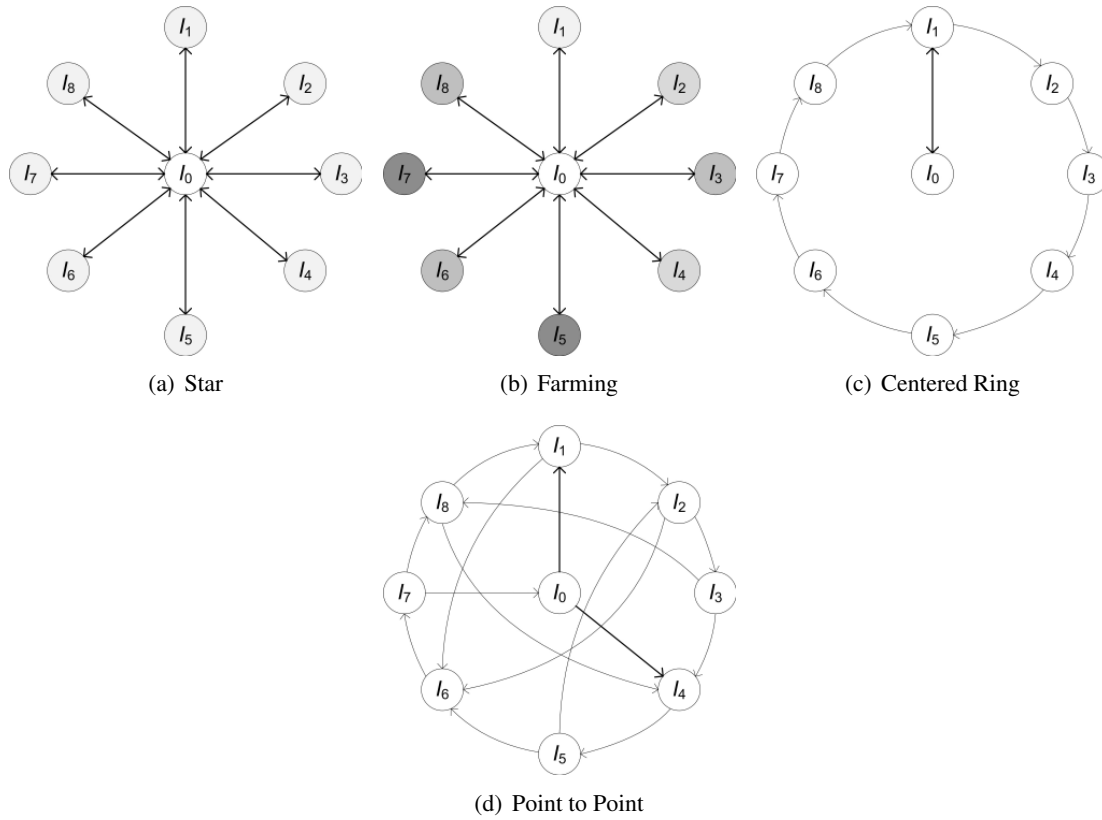


Figure 3.3: Application topologies

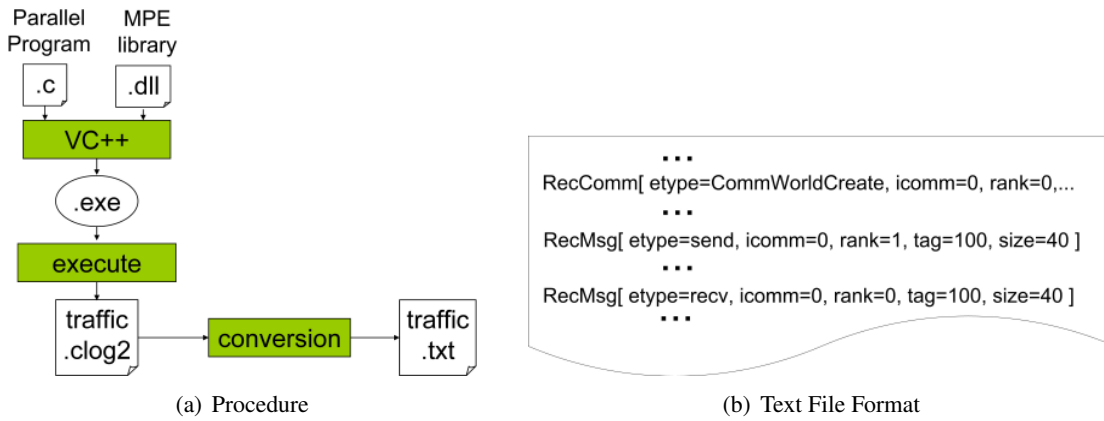


Figure 3.4: Extracting application traffic from simulations

Table 3.4: Parameters of used processors

| Name | t_j | A_j (slices) | Prog. mem size (MB) |
|------------|-------|----------------|---------------------|
| microblaze | 0.70 | 480 | 64 |
| PowerPC | 0.625 | 0 | 16 |

need to be saved during a context switch. For microblaze, it was assumed that all 32 General Purpose and all writable 12 Special Purpose Registers (GPR and SPR) are saved and restored during each context switch. Because the SPRs need to be moved to the GPRs first before loading or storing, the resulting cycles for context switching are $2 \times (32+12) + 2 \times 12 = 112$. Similarly, for PowerPC, 32 GPRs and 18 writable SFRs need saving and restoration. However, because the SFRs can be directly loaded and stored, the resulting cycles for context switch are $2 \times (32+18) = 100$. The task switch time is given by

$$\frac{t_j}{T_{clock_j}} = \frac{\text{switch cycles}}{\text{clock frequency} * \text{scheduling interval}} \quad (3.33)$$

In each case, the clock frequency is arbitrarily set to 160MHz, and the interval is $1\mu s$. The interval is for a simple kernel performing round-robin scheduling. Because the PowerPC is a hard-core, the area is taken to be zero. All applications can fit into either program memory.

Because FSL is a direct link, the capacity M_k equals 2. For the same reason, both the arbitration probability p_k and bound τ_k are zero. OPB can support up to 16 processors. The arbitration probability and bound are given for the worst case when 16 processors share a bus. In contrast to FSL with its documented slice consumption, the size of OPB as used in this experiment is approximated for 16 processors. Since no adaptors are required, adaptor/bridge latencies $\mathcal{L}_{kj_1j_2}$ are zero so that the optimizations are conducted without constraints (3.28) and (3.29).

Up to 5 networks were used (i.e. $K \in \{1, 2, 3, 4\}$). Where the number of networks is even, the same number of OPB and FSL was used. Otherwise, the number of FSL used exceed that of OPB by one. Since both MB and PPC cores can be connected to either OPB or FSL, the parameters $V_{j_1j_2k}$ equal 2 in each combination so that the constraint (3.24) is left out from optimizations.

Given all of the preceding parameters, the formulator generates an ILP instance in the GNU MathProg format [80]. The ILP model is statically described in a MathProg file, whereas the model data is generated by the formulator into a MathProg data file during synthesis. The synthesis flow in Figure (3.1) uses the lp_solve library version 5.5 [81] to solve generated ILP instances. The synthesis was done on a machine with an Intel T5500 1.66GHz processor and 2048M of memory. Even though the processor has two cores, only one of them was utilized during optimizations, probably because lp_solve implementation is single-threaded. Experiments with several solver settings were conducted, of which a combination of “most feasible basis crash” and the “ordering” of branch & bound variables [81] appear to significantly speed-up the solving of the proposed ILP formulation. The “most feasible basis crash” is a heuristic used to select the initial basis matrix for the simplex algorithm[81].

Because the size of the problem and hence the feasibility of the synthesis is determined by the number of processors, networks and tasks, three different runs of experiments were conducted, where two of the three parameters were fixed, and the other varied. Initially, the number of processors in the processor set was increased from 2 to 16, starting with a MB and a PPC, adding a further MB and a PPC, and adding 2 MBs in subsequent scenarios. The number of tasks and of networks was fixed at 4 and 2 respectively (1 OPB and 1 FSL). In the second run, the number of networks was increased to 5, while using 16 processors. Finally, the number of tasks was stepwise increased to 22 while using 16 processors and 5 networks. In total, synthesis was conducted for 117 different scenarios.

Table 3.6 summarizes application characteristics in terms of data traffic (total volume in KBytes and number of transfers), and in terms of the number of edges E as obtained from MPI simulations

Table 3.5: Parameters of used networks

| Type | M_k | L_k (μs) | p_k | τ_k (μs) | A_k (slices) |
|------|-------|-------------------|--------|----------------------|----------------|
| link | 2 | 6.25 | 0 | 0 | 451 |
| bus | 16 | 8.00 | 0.0625 | 16 | 451 |

with different number of tasks N . As previously noted, synthesis for the Jacobi implementation was conducted for 4 tasks only because of its non-scalable point-to-point topology. Figure 3.5 shows the estimated run time of tasks on MB for scalable implementations for different number of tasks.

A qualitative inspection of the table and of the figure reveals that synthesized architectures will tend to have a few number of processors to minimize communication cost, i.e. solutions use fewer processors than available to minimize communication costs.

FIR implementation has a constant amount of total data transferred, but the run time of individual tasks decay with higher degree of parallelism. As a result, inter-task communication becomes more expensive with a higher number of tasks. The situation is the same for derivation even though the runtime does not decay as fast. For the other three implementations, the data traffic increases with a higher degree of parallelism, while the runtime of tasks decay as seen in figure 3.5, thus leading to a much steeper increase in inter-task communications. Table 3.8 which shows synthesized architectures confirm this conclusion.

The implication of this result is that a designer maybe able to reduce synthesis runtime during explorations with different number of tasks by comparing the traffic against task runtime. By so doing, the number of processors in the IP library can be kept small, thereby reducing the complexity of the ILP instance. The usefulness of the reduction is apparent in Figure 3.6 which shows the solver CPU time when the number of processors in the IP library is increased. A general exponential increase can be observed for all six parallel programs. Such a reduction may however not be possible if all processors were different since that would limit the design space.

In contrast to processors, the number of networks should not be reduced because inter-task communication can be reduced by deploying more networks. That is evident from synthesized architectures, where the optimization tends to allocate several networks even when the number of allocated processors is only two. Surprisingly though, the number of networks appears not to have a significant influence on solver runtime as is evident in Figure 3.7. This may however not be the general case.

Figure 3.8 shows the solver runtime in log scale as the number of tasks is increased. As we would expect from Tables 3.1 and 3.2, the solver runtime increases significantly with a large number of tasks since the problem size increases non-linearly. In this run, an arbitrarily selected timeout of 28800 seconds was set to limit the solver runtime. In the figure, the 9 columns which exceed the time limit represent runs which did not find a feasible solution by timeout. Columns which terminate at timeout value represents runs for which feasible solutions were found, but the search was not completed so that synthesized architectures are possibly suboptimum. Therefore, it is interesting to examine the quality of those results. Figure 3.9 shows the gaps in percentage of the found solutions over relaxed solutions⁴. The gap is calculated by evaluating the objective function, and is given as

$$\text{Gap} = \frac{|s_i - s_r|}{s_r} \quad (3.34)$$

where s_r is the value of the objective function after evaluating a relaxed solution, and s_i the value obtained by evaluating a feasible (or optimum) integer solution.

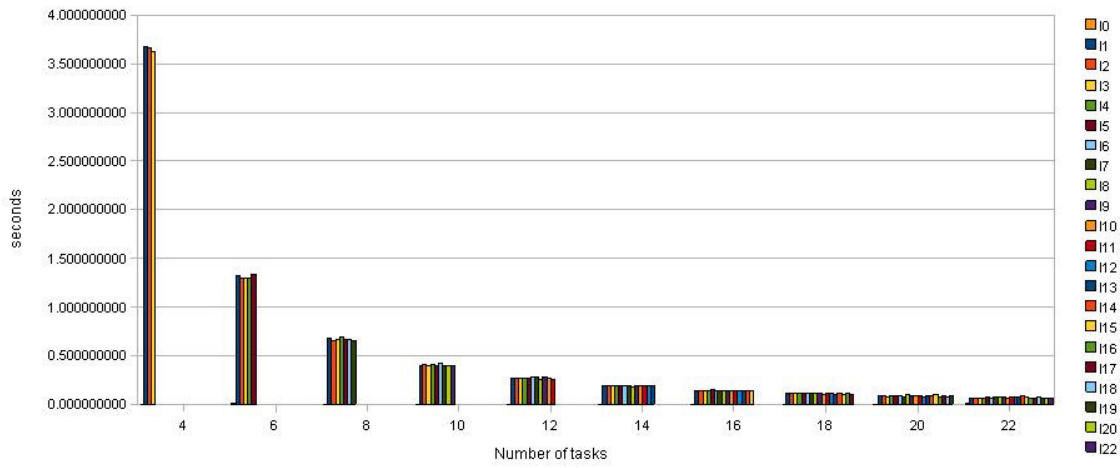
The figure shows that the gaps of feasible results obtained before the completion of search were not significantly large in every case. For instance, suboptimum architectures were obtained for Simpsons for $14 \leq N \leq 22$, but the gap was only 0.1%. Also, for Derivation, gaps obtained for $12 \leq N \leq 20$ are approximately the same, even though there was a timeout for $N = 18$ and $N = 24$. This result indicate that useful architectures can be obtained even under solver timeout. The gaps give a measure of the usefulness of the results in such cases.

Moreover, FIR results demonstrate that the combinatorial nature of the problem can lead to solutions for which the value of the objective is significantly large compared to that of the relaxed solution. This can potentially offset the benefits of additional number of tasks: when a higher degree of parallelism

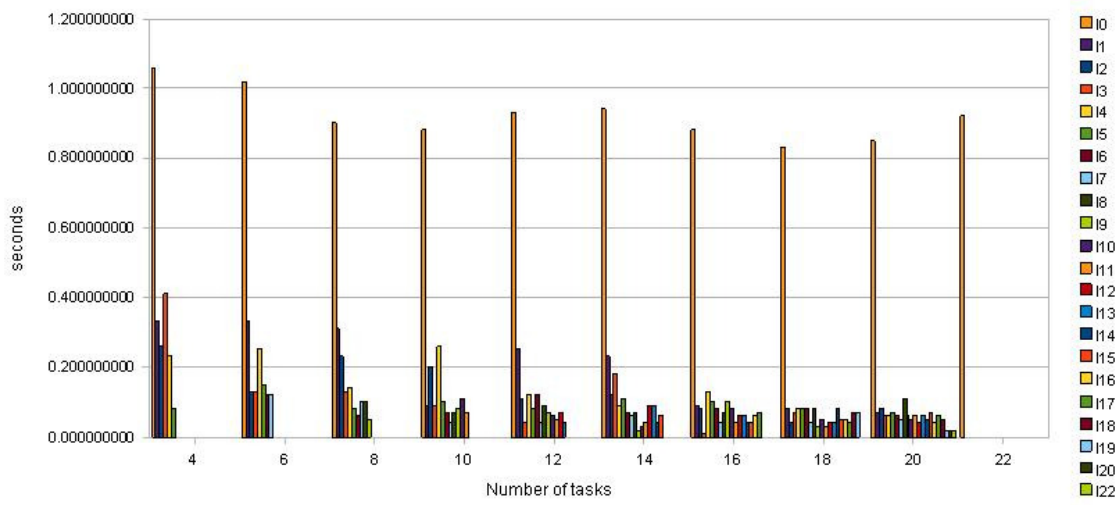
⁴Relaxed solution in the context of Branch & Bound algorithm.

Table 3.6: A summary of application characteristics

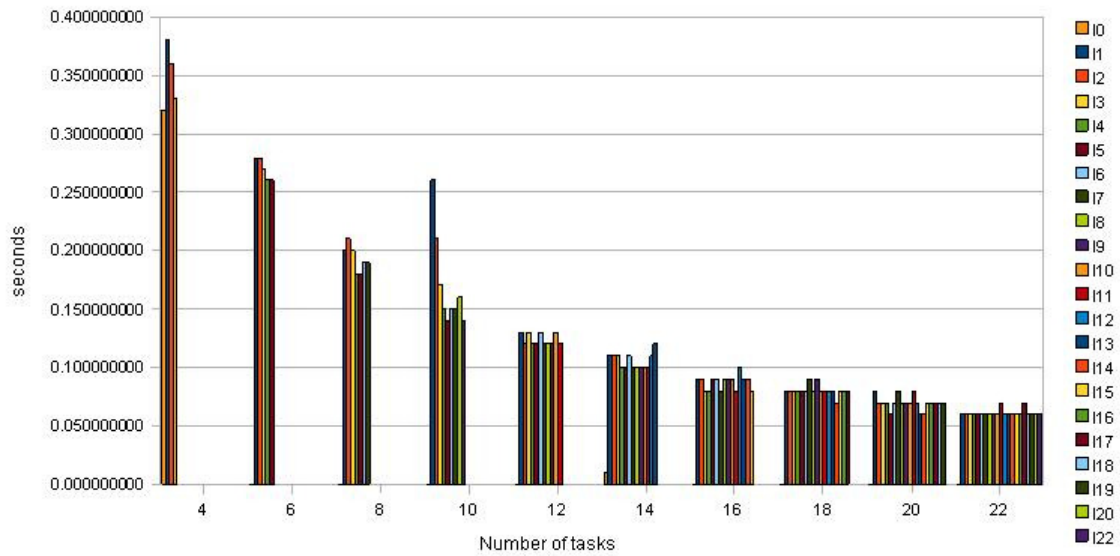
| N | FIR | | | Simpsons | | | N-Body | | | Inversion | | | Derivation | | |
|---------------|-----|--------|-------------|----------|-------|-------------|--------|--------|-------------|-----------|--------|-------------|------------|--------|-------------|
| | E | KBytes | # Transfers | E | Bytes | # Transfers | E | MBytes | # Transfers | E | MBytes | # Transfers | E | KBytes | # Transfers |
| 4 | 6 | 781 | 9 | 6 | 72 | 6 | 13 | 4 | 120 | 9 | 736 | 81944 | 6 | 27 | 1001 |
| 6 | 10 | 781 | 15 | 10 | 120 | 10 | 35 | 6 | 300 | 19 | 1120 | 135208 | 10 | 27 | 1003 |
| 8 | 14 | 781 | 21 | 14 | 168 | 14 | 58 | 9 | 560 | 30 | 1505 | 188472 | 14 | 27 | 1005 |
| 10 | 18 | 781 | 27 | 18 | 216 | 18 | 99 | 11 | 900 | 42 | 1889 | 241736 | 18 | 28 | 1007 |
| 12 | 22 | 781 | 33 | 22 | 264 | 22 | 144 | 13 | 1320 | 52 | 2273 | 295000 | 22 | 28 | 1009 |
| 14 | 26 | 781 | 39 | 26 | 312 | 26 | 195 | 16 | 1820 | 65 | 2657 | 348264 | 26 | 28 | 1011 |
| 16 | 30 | 781 | 45 | 30 | 360 | 30 | 256 | 18 | 2400 | 79 | 3041 | 401528 | 30 | 28 | 1013 |
| 18 | 34 | 781 | 51 | 34 | 408 | 34 | 324 | 21 | 3060 | 97 | 3425 | 454792 | 34 | 28 | 1015 |
| 20 | 38 | 781 | 57 | 38 | 456 | 38 | 387 | 23 | 3800 | 111 | 3809 | 508056 | 38 | 28 | 1017 |
| 22 | 42 | 781 | 63 | 42 | 504 | 42 | 450 | 25 | 4540 | 176 | 4194 | 561320 | 42 | 28 | 1019 |
| Jacobi | | | | | | | | | | | | | | | |
| 4 | 14 | 40 | 426 | | | | | | | | | | | | |



(a) FIR

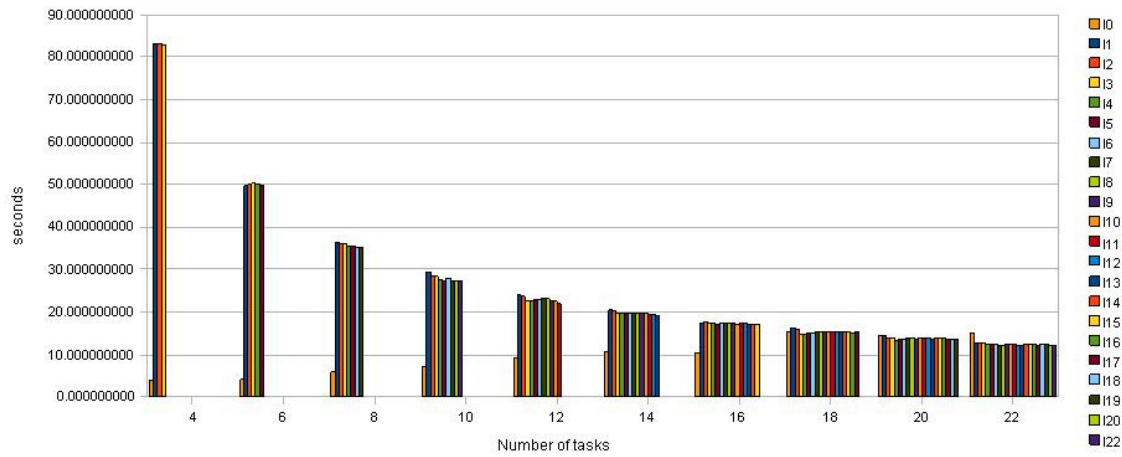


(b) Derivation

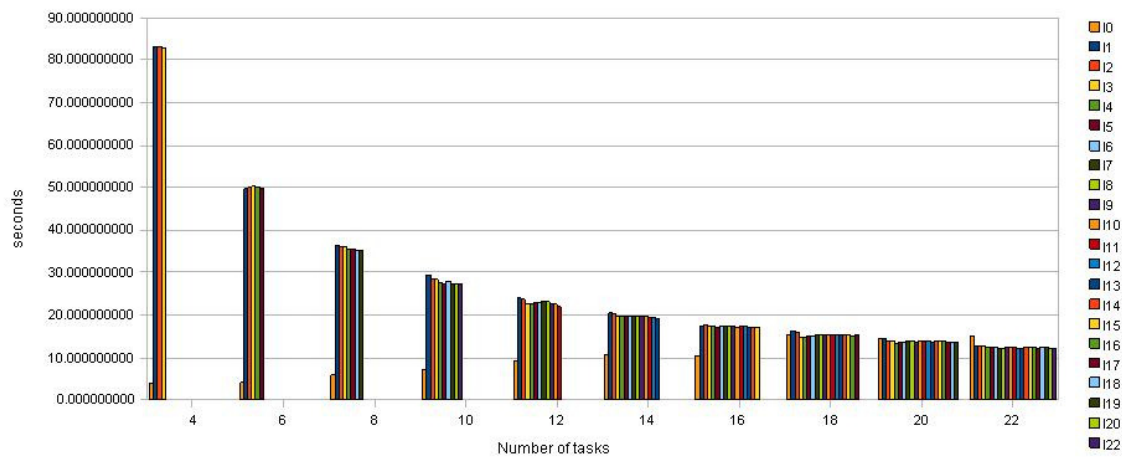


(c) Simpsons

Figure 3.5: Continued on the next page



(a) Inversion



(b) N-Body

Figure 3.5: Estimated task run time on MB for different number of tasks

Table 3.7: Integer solutions (in seconds, suboptimum solutions in bold)

| # Tasks | FIR | Derivation | Simpsons | N-Body | Inversion |
|---------|-------------|-------------|-------------|--------------|-----------|
| 4 | 1.88 | 0.42 | 0.24 | 1.58 | 43.44 |
| 6 | 3.12 | 0.41 | 0.23 | 8.66 | 253.76 |
| 8 | 3.86 | - | - | 21.11 | 256.50 |
| 10 | 5.09 | 0.54 | 0.36 | - | 257.71 |
| 12 | 2.98 | 0.68 | 1.15 | - | 261.61 |
| 14 | 2.52 | 0.72 | 0.67 | 24.67 | 267.47 |
| 16 | 2.21 | 0.89 | 0.74 | - | 270.75 |
| 18 | 5.73 | 0.90 | 0.81 | - | - |
| 20 | 2.49 | 0.93 | 0.82 | - | - |
| 22 | - | 0.86 | - | - | - |

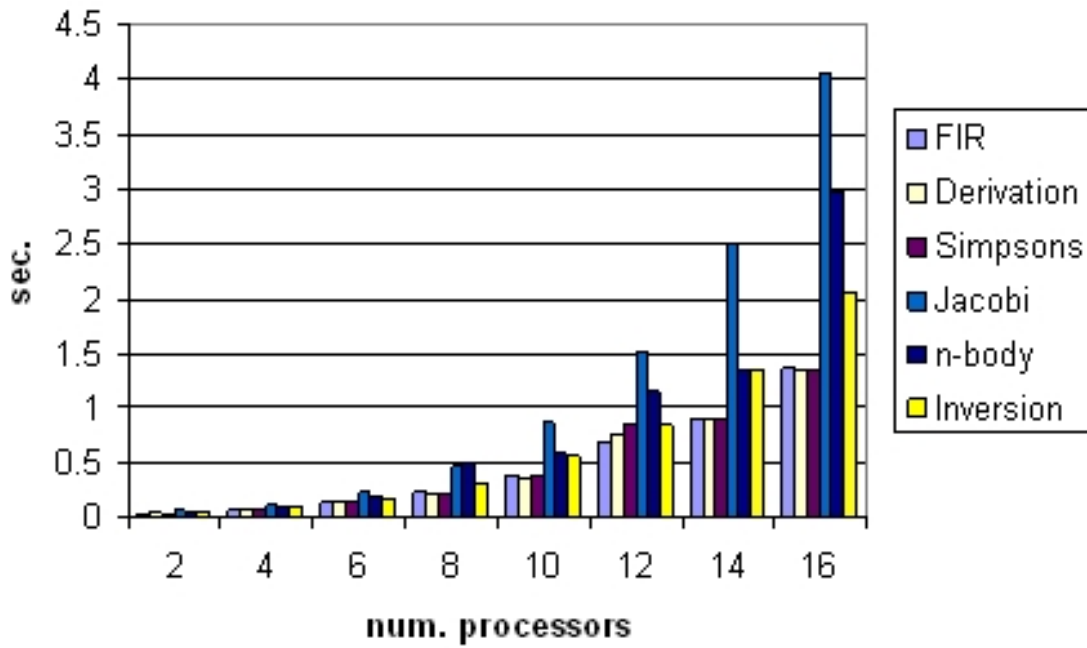


Figure 3.6: Solver runtime versus number of processors available

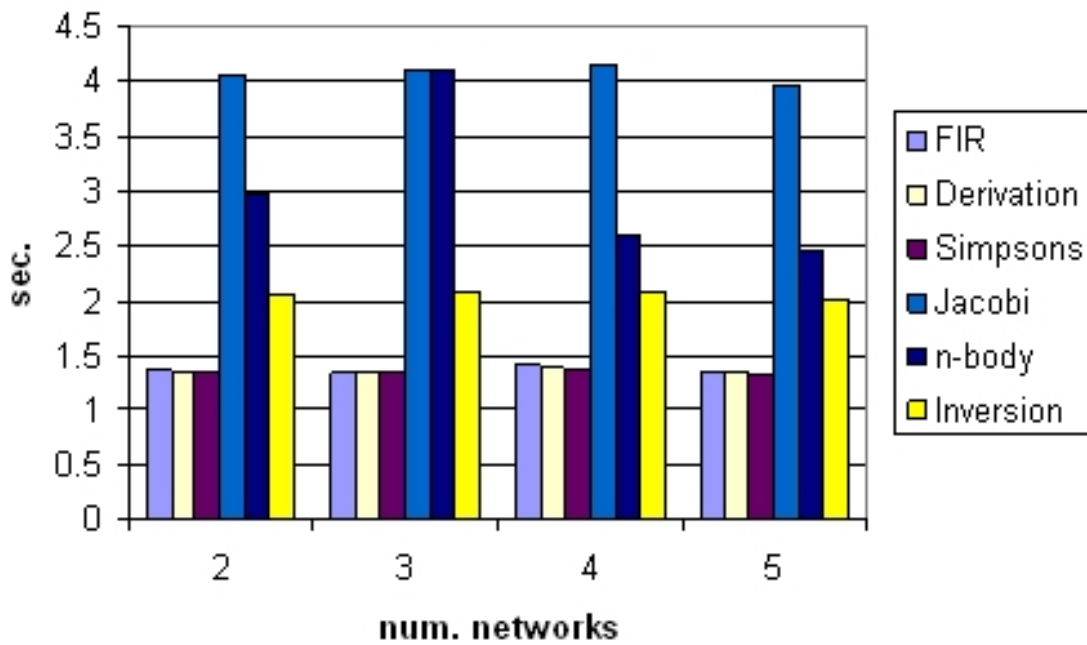


Figure 3.7: Solver runtime versus number of networks available

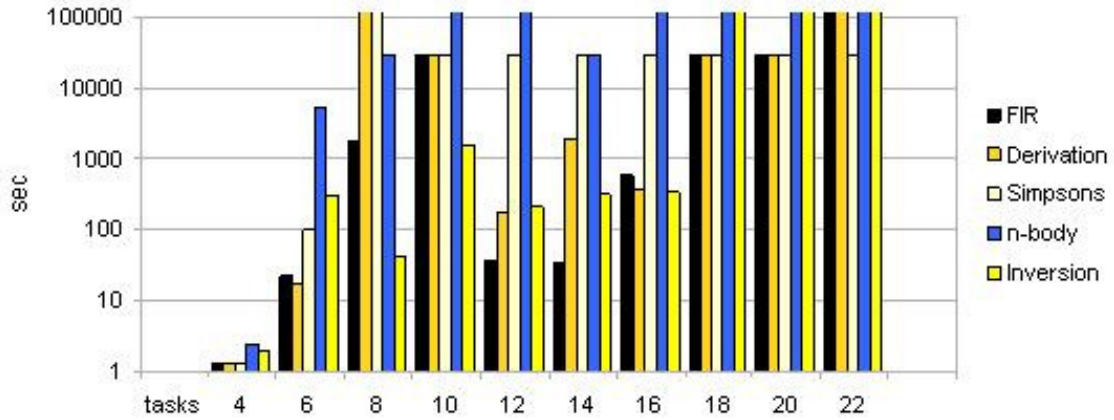


Figure 3.8: Solver runtime versus number of tasks

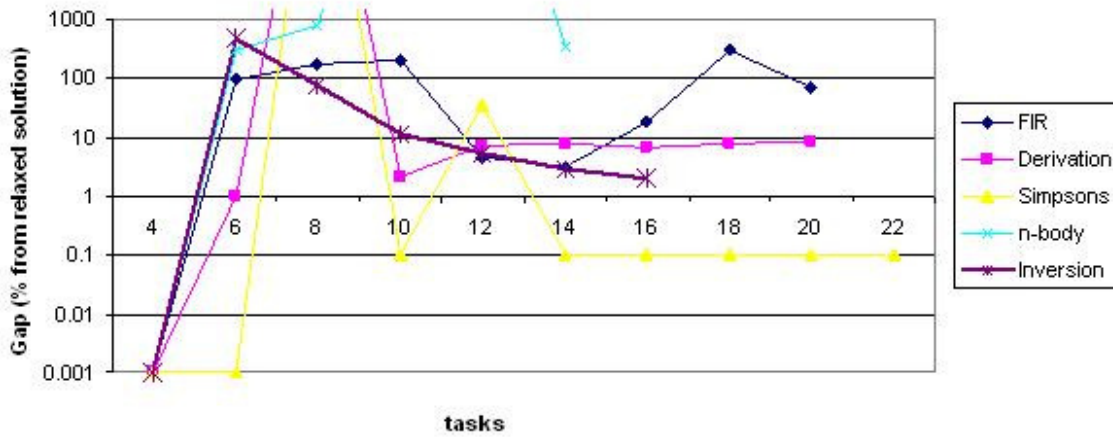


Figure 3.9: Gaps of found combinatorial solutions over relaxed solutions

is used so that the resulting relaxed solution is lower, then a sufficiently large gap indicates a much worse integer solution. For instance, in the FIR case, optimum architectures were obtained for $N = 6$ and for $N = 8$. However, the objective for the latter case worsened by 22% to 3.863 seconds (Table 3.7). The gaps for the two cases were 94% and 169% respectively. This behavior cannot be explained by the increase in inter-task communication and an increase in scheduling cost alone: the objective for $N = 12$ is less by 21% compared with $N = 8$, and the gap is only 4.5%, where, in contrast, all tasks are mapped on a single core.

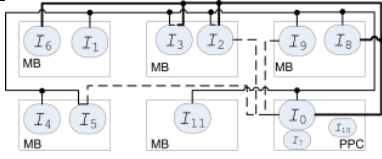
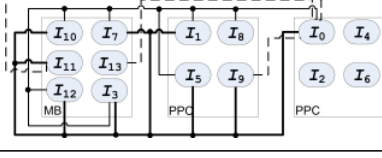
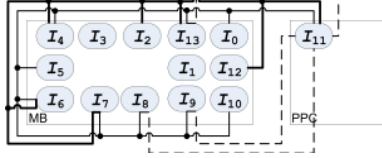
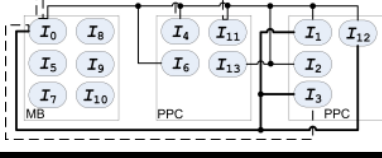
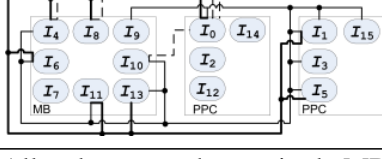
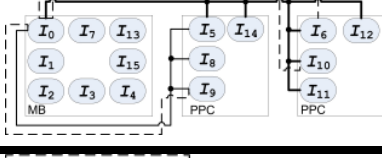
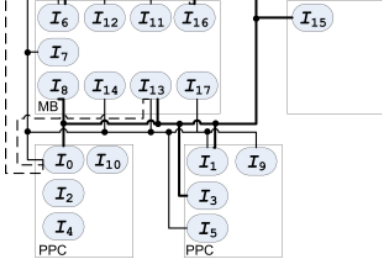
These counter intuitive results underscore the need for an automated synthesis to explore the design space as proposed in this thesis. Such trends are difficult to predict so that even a skilled designer can easily miss a vastly superior design point.

Table 3.8: Synthesized architectures for non-realtime parallel programs (time out = 28800 seconds; solid lines are buses, dashed lines are FSL)

| Parameters | | | Program | Architecture |
|--------------|-----|-----------------|-------------------|----------------------------------|
| K' | N | M | | |
| 2 | 4 | {2, 4, ..., 16} | All | All tasks mapped on a single PPC |
| {2, 3, 4, 5} | 4 | 16 | All | All tasks mapped on a single PPC |
| 5 | 6 | 16 | Inversion, N-Body | All tasks mapped on a single MB |
| | | | Derivation | |
| | | | Simpsons | |
| | | | FIR | |
| 5 | 8 | 16 | Inversion, N-Body | All tasks mapped on a single MB |
| | | | FIR | |
| 5 | 10 | 16 | Derivation | |
| | | | Inversion | All tasks mapped on a single MB |
| | | | FIR (suboptimum) | |
| | | | N-Body | Timed out |
| | | | Simpsons | |
| 5 | 12 | 16 | derivation | |
| | | | Inversion, FIR | All tasks mapped on a single MB |
| | | | N-Body | Timeout |

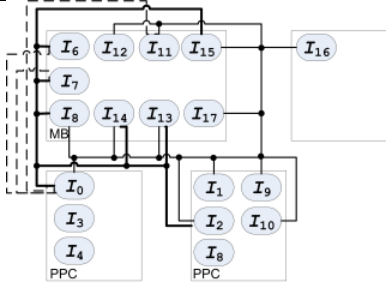
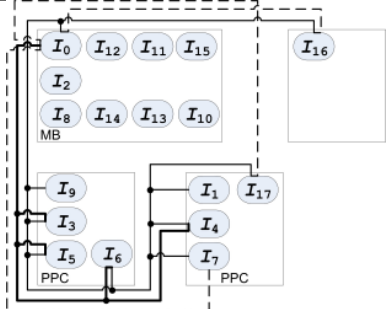
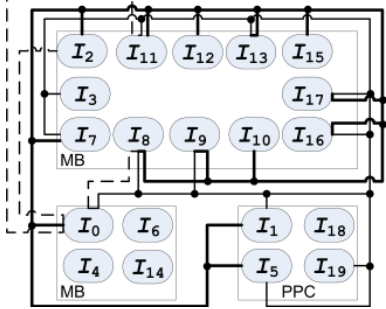
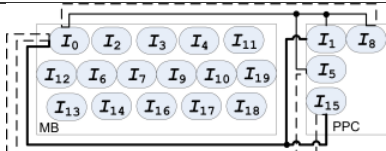
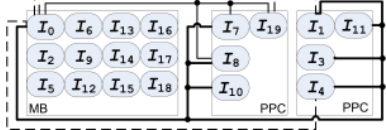
Continued on Next Page...

Table 3.8 – Continued

| K' | N | M | Program | Architecture |
|------|-----|-----|-------------------------|---|
| | | | Simpsons (suboptimum) |  |
| 5 | 14 | 16 | derivation |  |
| | | | Inversion, FIR | All tasks mapped on a single MB |
| | | | N-Body (suboptimum) |  |
| | | | Simpsons (suboptimum) |  |
| 5 | 16 | 16 | Derivation |  |
| | | | Inversion, FIR | All tasks mapped on a single MB |
| | | | N-Body | Timeout |
| | | | Simpsons (suboptimum) |  |
| 5 | 18 | 16 | Derivation (suboptimum) |  |
| | | | Inversion | Timeout |

Continued on Next Page...

Table 3.8 – Continued

| K' | N | M | Program | Architecture |
|------|-----|-----|-------------------------|--|
| | | | FIR (suboptimum) |  |
| | | | N-Body | Timeout |
| | | | Simpsons (suboptimum) |  |
| | | | Derivation (suboptimum) |  |
| 5 | 20 | 16 | Inversion | Timeout |
| | | | FIR (suboptimum) |  |
| | | | N-Body | Timeout |
| | | | Simpsons (suboptimum) |  |
| 5 | 22 | 16 | Derivation | Timeout |
| | | | Inversion | - |
| | | | FIR | - |
| | | | N-Body | - |

Continued on Next Page...

Table 3.8 – Continued

| K' | N | M | Program | Architecture |
|------|-----|-----|----------|--------------|
| | | | Simpsons | |

3.2.2 Scheduling

This section presents an extension to the ILP model to fully include scheduling effects, particularly, so as to synthesize architectures for realtime applications. As discussed in Section 2.2.2, only local schedules are considered because almost always preemption is the preferred method. The extension involves replacing the switching time (3.5) by

$$T_{switch} = \sum_{l=0}^{2^n-2} \sum_{j=0}^m \gamma_{lj} \cdot F_{lj} \cdot (T_{s_{lj}} \cdot t_j + O_j) \quad (3.35)$$

This model captures influences from several design parameters, each of which is described in a subsection below.

3.2.2.1 The Processor Architecture

This is the actual cost of switching context, which depends on the memory and on the micro architecture, as well as on the mechanism for context switching (i.e. under software or with hardware support). The coefficient t_j captures this cost, and can always be reliably pre-computed for processors of interest. Whether this cost is actually incurred, depends on task mapping as discussed in Section 3.2.2.4.

3.2.2.2 The Kernel/OS

The kernel or realtime OS introduces a control overhead due to scheduling (polling, moving tasks between run and delay ques, etc.). In this formulation, it is assumed that kernel/OS is already selected and is fixed for each of the processor in the design space (i.e. OS selection is not a part of the optimization problem so that the associated cost is coupled to the selected processor). This is not a limitation because when desired, instances of the same processor running different operating systems or micro-kernels can be specified in the ILP problem to extend the design space.

The overhead is caused by the clock interrupt handler interfering with the execution of application tasks because of its higher priority. This increases the number of task switching and the response time of application tasks. The coefficient O_j in (3.35) captures the increase in response time caused by the clock-handler.

3.2.2.3 The Schedule/Task Switching

The schedule determines how often task switching takes place as captured by the coefficient $T_{s_{lj}}$ in (3.5). $T_{s_{lj}}$ is the number of task switching that is incurred for the duration of the application, or for one period, when a particular group of tasks with the index l is mapped on a processor J .

In simple cooperative schedulers (e.g. cyclic executives), there is no preemption, so that $T_{s_{lj}} = 0$. The overhead incurred when a task begins and ends is already captured by T_{ij} in (3.31) as part of function call overhead (prologue and epilogue). For preemptive schedulers, $T_{s_{lj}}$ is equal to the number of interferences due to higher priority tasks. Section 3.2.2.6 discusses how the parameter is obtained.

3.2.2.4 Task Mapping

Task mapping influence the switching costs in two ways : (1) by selecting the processor, the switching mechanism, and thus the cost, is determined, and (2) by grouping tasks on one processor, the optimum schedule that can be applied, and thus the number of task switching, is determined. Consequently, scheduling and task-mapping influence each other during optimization.

To include this cross-effect during optimization, two strategies can be followed:

- i Integrate scheduling in optimization loop such that a schedule is computed prior to cost calculation for a candidate mapping. Applicable where the optimizer is an integral part of the synthesis flow. This approach is used in Chapter 5.
- ii Pre-compute optimum schedules for all possible mappings, and integrate the schedules in the ILP formulation. Applicable where the solver is an independent module as is the case in this ILP-based synthesis. This approach has the advantage that, by pre-computing the schedules, infeasible mappings can be eliminated to reduce the size of the ILP instance. The drawback is the explosion of variables when all task combinations are feasible on every processor. Pre-computation is used in this and in Chapter 4.

The coefficient F_{lj} in (3.5) specifies feasible mappings following pre-computed schedules. Its value is 1 if there is a feasible schedule for a group of tasks with the index l on processor J , 0 otherwise. We next describe how F_{lj} is used to enforce feasibility constraints in the formulation.

Let $\mathcal{P}(I)$ be a power set of the task sets $I = \{I_0, \dots, I_n\}$. Let G_l be an element in the power set excluding the empty set, with $l = \{0, 1, \dots, 2^n - 2\}$. Let $I_{mapped} \subset \mathcal{P}(I)$ be a set, so that each element contain one or more tasks that will be mapped on the same processor. The solution to the combinatorial optimization problem consists of the set I_{mapped} . Each element in I_{mapped} is associated with a task switching overhead as dictated by its schedule.

Now since I_{mapped} is not known at formulation time, an auxiliary binary decision variable \mathcal{M}_{lj} is introduced for each element G_l in the power set $\mathcal{P}(I)$. If $\mathcal{M}_{lj} = 1$, then $G_l \in \mathcal{P}(I)$, and $G_l \in I_{mapped}$. If $\mathcal{M}_{lj} = 0$, then G_l is not an element of I_{mapped} . Therefore, we insist that

$$\mathcal{M}_{lj} \leq F_{lj} \quad \forall \mathcal{M}_{lj} \quad (3.36)$$

so that if, and only if, the mapping is feasible, then \mathcal{M}_{lj} constitutes a degree of freedom during synthesis where the variable can take any value. We next describe how the decision variables \mathcal{M}_{lj} and x_{ij} are linked through ILP constraints.

Recalling that $x_{ij} = 1$ implies that a task I_i is mapped on a processor J_j , it follows for any group G_l , $\mathcal{M}_{lj} = 1$ if and only if $x_{ij} = 1 \quad \forall I_i \in G_l$. This results into a logical constraint

$$\begin{aligned} \mathcal{M}_{lj} &= (x_{i_{0j}} \wedge x_{i_{1j}} \wedge \dots \wedge x_{i_{lgj}}) \quad \forall \mathcal{M}_{lj} \text{ with} \\ G_l &= \{I_{i_0}, I_{i_1}, \dots, I_{i_g}\}, \quad lg = |G_l| - 1 \end{aligned} \quad (3.37)$$

To transform the logical constraint into a linear form, two steps are applied. The first is the specification

$$\begin{aligned} \mathcal{M}_{lj} = 0 &\rightarrow (x_{i_{0j}} + x_{i_{1j}} + \dots + x_{i_{lgj}}) < |G_l| \\ \mathcal{M}_{lj} = 1 &\rightarrow (x_{i_{0j}} + x_{i_{1j}} + \dots + x_{i_{lgj}}) = |G_l| \end{aligned} \quad (3.38)$$

These two constraints insure that when a schedule is not feasible, then at least one $I_i \in G_l$ is not be mapped on J_j . This implies that other groups which are either proper subsets of G_l , or which are not super sets of G_l , can be mapped on J_j provided they have a feasible schedule. The second step then is a set of inequalities that satisfy the specification in (3.38)

$$x_{i_{0j}} + x_{i_{1j}} + \dots + x_{i_{lgj}} \leq |G_l| - 1 + \mathcal{M}_{lj} \quad (3.39)$$

$$x_{i_{0j}} + x_{i_{1j}} + \dots + x_{i_{lgj}} \geq |G_l| \cdot \mathcal{M}_{lj} \quad (3.40)$$

With (3.36), (3.39) and (3.40), feasible mappings are guaranteed. The last step is to capture the switching cost of the groups in the objective. A contribution of a group to the switching cost is given by

$\mathcal{M}_{l_j} \cdot T_{s_{l_j}}$. However, this contribution cannot be directly used in the objective function by taking the sum of all contributions from all groups. This is because, as it can be observed from (3.39) and (3.40), if a group G_l is mapped on a processor J_j , then the value of \mathcal{M}_{l_j} for all groups which are subsets of G_l will also be one. Consequently, taking the sum of contributions directly would erroneously include the switching cost of subgroups.

To prevent this incorrect inclusion of switching costs, we need to specify that the switching contribution of a group should be counted only when $\mathcal{M}_{l_j} = 0$ for all of its super sets. This leads to non-linear terms in the objective function which take the form $\mathcal{M}_{l_j}(1 - \mathcal{M}_{l_{s1j}})(1 - \mathcal{M}_{l_{s2j}}) \cdots$, where l_{s1} and l_{s2} are indices of super sets for a group with an index l .

To break the non-linearity, a binary decision variable γ_{l_j} is introduced for each \mathcal{M}_{l_j} . Its value is 1 only when the group is mapped, and $\mathcal{M}_{l_j} = 0$ for all super sets. To model this property, we note from (3.40) that the LHS of the inequality for a group G_{l_1} with $\mathcal{M}_{l_1j} = 1$ is greater than the LHS of the same inequality for a group G_{l_2} , if $G_{l_2} \subset G_{l_1}$. Therefore, the following relationship holds:

$$0 \leq \frac{1}{|I|} \left(\sum_{I_i \in I} x_{ij} - \sum_{I_i \in G_l} x_{ij} \right) < 1 \quad \forall G_l, J_j \quad (3.41)$$

The first sum in the above relationship is the total number of tasks that have been mapped on J_j . The second sum is the size of a group, which is the same as the LHS of (3.40). The difference of the two sums is zero in two cases (1) if nothing is mapped on J_j (2) if a mapped group has no superset G_{l_s} for which $\mathcal{M}_{l_sj} = 1$ for that specific mapping. The sum is greater than zero, if, for G_l , there is a superset G_{l_s} with $\mathcal{M}_{l_sj} = 1$. This is because there is then at least one decision variable x_{ij} with value 1 in the first sum, which is not present in the second. The largest value that the difference of the two sums can have is $|I| - 1$, so that the upper limit in (3.41) is 1.

We next exploit this relationship by specifying that

$$\gamma_{l_j} + \frac{1}{|I|} \left(\sum_{I_i \in I} x_{ij} - \sum_{I_i \in G_l} x_{ij} \right) \leq 1 \quad (3.42)$$

$$\gamma_{l_j} + \frac{1}{|I|} \left(\sum_{I_i \in I} x_{ij} - \sum_{I_i \in G_l} x_{ij} \right) \geq \mathcal{M}_{l_j} - 1 + \frac{1}{|I|} \quad (3.43)$$

Table 3.9 shows that the decision variable γ_{l_j} can only take the value 1 when:

1. No grouped is mapped on J_j (line 1 in the table). In that case, optimizations will force the value 0 to minimize T_{switch} in (3.35).
2. When G_l has no super set mapped on J_j (line 2), since the variable will assume the more constraining value.

3.2.2.5 Processor-External Factors

Processor-external factors such as interrupts and data availability have a direct runtime impact on the schedule. The foregoing formulation has the limitation that it is based on worst case assumptions. In

Table 3.9: Truth table for inequalities (3.42) and (3.43)

| \mathcal{M}_{l_j} | weighted diff. | γ_{l_j} in eq.(3.42) | γ_{l_j} in eq.(3.43) |
|---------------------|----------------|-----------------------------|-----------------------------|
| 0 | 0 | 0,1 | 0,1 |
| 1 | 0 | 0,1 | 1 |
| 1 | > 0 | 0 | 0,1 |

particular, it is assumed that tasks in a group G_l with no precedence relationship can become ready at the same time.

With respect to data availability, the worst case assumptions can be relaxed by considering when data can actually arrive depending on source-task-destination-task mapping, and depending on the selected communication network.

A possible relaxing solution is to compute offsets between release times of tasks with indirect precedence. For example, if there are three tasks such that the first sends data to the second, and the second to the third, and the mapping is such that the first and third are mapped on one processor, and the second on another, then there is an offset between release times of the first and third tasks. This offset is equal to the time needed for data to be sent to the second task, plus the response time of the second task, plus the time for the resulting data to be sent from the second to the third task. A suitable ILP formulation that will not significantly increase the problem size needs to be found in future work.

3.2.2.6 Experimental Results

This section presents experimental results aimed at studying the feasibility of ILP-based synthesis after extending the scheduling aspect of the model. Also in this case, selected applications exhibit the same communication pattern for the same input data, but the individual tasks have finishing deadlines requiring realtime scheduling. Again, the parallelization style uses the MPI standard.

Two applications were used. The first implements a signal processing chain for IEEE 802.11g WLAN standard. The implementation performs, in the order, the following : timing synchronization, coarse and fine frequency offset estimation and compensation, symbol demapping, FFT (OFDM demodulation), pilot extraction, channel estimation and compensation, carrier phase offset estimation and compensation, and finally timing drift estimation and compensation. These algorithms are described in [82, 83, 84, 85]. The second application implements baseband signal processing for WCDMA, and performs, in the order, the following: matched filtering in frequency domain (using overlap-add), PN code tracking using a coherent Delay Locked Loop (DLL), channel estimation and compensation, descrambling and despreading, and finally maximum ratio combining. These algorithms are described in [86, 87, 88].

The applications were first written in sequential C, and were then converted into non-scalable MPI programs with point-to-point topology. Tables 3.10 and 3.11 show the parallel tasks and their deadlines in nanoseconds. In this implementation, the deadlines are equal to the periods. The latter were obtained from the standards. The three last columns in Table 3.10 show approximated execution time in nanoseconds of the tasks on 3 different processors with loosely coupled accelerators. Similarly, the two last columns in Table 3.11 show the approximated time for WCDMA tasks on two other processors.

Table 3.10: WLAN tasks(all time units in ns)

| Function | Index | Deadline | P1 | P2 | P3 |
|---|-------|----------|-------|------|-----|
| Master control task | I0 | 19 | 6032 | 1034 | 11 |
| Carrier phase offset est. & compensation | I1 | 988 | 3504 | 601 | 601 |
| Channel estimation and compensation | I2 | 988 | 909 | 156 | 156 |
| Timing synchronization | I3 | 19 | 33781 | 5791 | 17 |
| OFDM symbol demapping | I4 | 1216 | 598 | 102 | 102 |
| FFT (OFDM demodulation) | I5 | 1216 | 534 | 534 | 534 |
| Fine freq. offset estimation & compensation | I6 | 19 | 11 | 11 | 11 |
| Coarse freq. offset estimation & compensation | I7 | 19 | 13 | 13 | 13 |

Table 3.11: WCDMA tasks (all time units in ns)

| Function | Index | Deadline | P4 | P5 |
|--|-------|----------|-------|------|
| Master control task | I0 | 65 | 4376 | 43 |
| FFT (frequency domain filtering) | I1 | 4160 | 3788 | 3788 |
| Vector multiplication (freq. domain filtering) | I2 | 4160 | 3082 | 3082 |
| FFT (frequency domain filtering) | I3 | 4160 | 3788 | 3788 |
| Overlap-Add (frequency domain filtering) | I4 | 4160 | 3665 | 3665 |
| Long code generation | I5 | 1040 | 3784 | 147 |
| DLL (delay accusation & tracking) | I6 | 1040 | 8746 | 1000 |
| Short code generation | I7 | 1040 | 835 | 143 |
| Rake receiver | I8 | 1040 | 16204 | 686 |
| Maximum ratio combiner | I9 | 16640 | 383 | 383 |

Tables 3.12 and 3.13 show the communication traffic between tasks for the two applications. These were obtained from MPI simulations for 27 OFDM symbols for WLAN, and for 1 slot for WCDMA. The third and fourth columns show the amount and number of data transfers respectively. Also in this case, comparing the estimated execution time of tasks on processors with the data traffic, it become apparent that inter-task communications are expensive so that processors sharing is favored.

The same networks were used as in the previous section, however the number of instances was selected to be 12 for each type to cater for more number of edges. Table 3.14 show the parameters of used processors which are based on MB with loosely coupled accelerators. This selection reflect the fact that these tasks are very data intensive so that highly specialized processor cores are required. The use of less specialized processors is possible through the use of finer grained tasks. However, that option was not pursued in this experiment because FPGA slice estimates for the MB suggests that only a maximum of 16 cores can fit in the target Virtex 2 FPGA.

Since it is interesting to see the impact of scheduling constraints, synthesis was conducted with and without fixed-priority preemptive scheduling by using two configurations with a basic cyclic executive and with a preemptive kernel respectively. For the cyclic executive, the switching overhead due to preemption is zero.

For fixed-priority preemptive schedulers, the number of context switching T_{slj} is equal to the number of interferences due to higher priority tasks, and is obtained from Rate Monotonic Analysis (RMA)

Table 3.12: Traffic pattern for WLAN for 27 OFDM symbols

| Src (i_1) | Dst (i_2) | $D_{i_1 i_2}$ (bytes) | $B_{i_1 i_2}$ |
|---------------|---------------|-----------------------|---------------|
| I0 | I2 | 220 | 55 |
| I0 | I7 | 2339608 | 2242 |
| I0 | I1 | 220 | 55 |
| I4 | I2 | 5824 | 28 |
| I4 | I1 | 5824 | 28 |
| I6 | I5 | 7168 | 28 |
| I2 | I1 | 5824 | 28 |
| I7 | I6 | 2913300 | 2241 |
| I5 | I4 | 7168 | 28 |
| I1 | I3 | 5824 | 56 |

Table 3.13: Traffic pattern for WCDMA for one slot

| Src (i_1) | Dst (i_2) | $D_{i_1 i_2}$ (bytes) | $B_{i_1 i_2}$ |
|---------------|---------------|-----------------------|---------------|
| I8 | I9 | 40948 | 2560 |
| I1 | I2 | 81924 | 161 |
| I0 | I1 | 81412 | 160 |
| I5 | I8 | 10236 | 2559 |
| I5 | I6 | 40960 | 10240 |
| I4 | I6 | 40964 | 10241 |
| I3 | I4 | 81924 | 161 |
| I2 | I3 | 81924 | 161 |
| I6 | I8 | 81892 | 5119 |
| I7 | I8 | 10236 | 2559 |

Table 3.14: Parameters of used processors

| Name | t_j (μs) | A_j (slices) | # instances |
|------|-------------------|----------------|-------------|
| P1 | 0.70 | 450 | 5 |
| P2 | 0.93 | 526 | 6 |
| P3 | 1.20 | 605 | 8 |
| P4 | 1.20 | 553 | 6 |
| P5 | 1.19 | 724 | 7 |

[89] within the ILP formulator (Figure 3.1). The RMA in the ILP formulator currently support tasks with single deadlines, and which have fixed durations and non-varying periods. However, the implementation is easily extensible to support flexible RMA models. Such models can be adapted to applications with arbitrary, multiple or internal deadlines [90]. Future extensions will affect the computation of $T_{s_{ij}}$ only. The ILP model for synthesis remains unaffected.

In the implemented flow, RMA is conducted for all possible task groups and mappings. The output of RMA, the response r , is used to estimate $T_{s_{ij}}$. Algorithm 1 shows how the parameter is computed.

- 1: Table $T = \text{createSchedulingTable}(G_l, J_j, \text{task time } T_{ij}, \text{task deadlines } D_i, \text{task periods } T_i)$
- 2: **for all** $I_i \in G_l$ **do**
- 3: Response $r_i = \text{computeResponse}(I_i, T)$
- 4: $F_{lj} = 1$
- 5: **if** $r > D_i$ **then**
- 6: $F_{lj} = 0$
- 7: **end if**
- 8: **end for**
- 9: $r = \text{computeLargestResponse}(G_l, T)$
- 10: $T_{s_{ij}} = \left\lceil \frac{r}{\text{period of highest priority task in } G_l} \right\rceil$

Algorithm 1: Determining context switching cost using RMA

The first line computes a scheduling table for a group G_l of tasks, if the group would be mapped on a processor J_j . The rows in the scheduling table contain the priority of a task in the group, together with its deadlines, period and execution time on the processor, one row for each task. The priorities are computed according to [89].

In this implementation, the table is initially filled in arbitrary order with task information, but the priorities are initially zero. The rows are then sorted in two passes according to periods and deadlines. Prior to sorting, deadlines are relaxed according to Algorithm 2 in order to avoid pessimistic schedulability analysis. The analysis assumes that all tasks are released at the same time. If the response of a task is then greater than the deadline as shown in Algorithm 1, the schedule is declared infeasible. However, if there is a precedence relationship, not all tasks are released at the same time. In particular, if there is an edge between I_{i_1} and I_{i_2} , then I_{i_2} cannot start until I_{i_1} has finished. Therefore, the deadline D_{i_2} needs to be relaxed to $D_{i_2} = D_{i_2} + D_{i_1}$ to reflect the fact that there is an offset from the release time of its parent task.

```

1: For each  $I_i \in G_l$ , set parent deadline to zero
2: while  $G_l$  is not empty do
3:   if  $G_l$  is circular then
4:     select  $I_i \in G_l \mid \forall I_{i_2} \in G_l, \mathcal{T}_i < \mathcal{T}_{i_2}$ 
5:   else
6:     select  $I_i \in G_l$  with no parent in  $G_l$ 
7:   end if
8:   for each child  $I_{i_2}$  of  $I_i$  in  $G_l$  do
9:     if parent deadline  $< D_i$  then
10:      set parent deadline to  $D_i$ 
11:    end if
12:  end for
13:  if  $I_i$  has a parent in  $G_l$  then
14:     $D_{i+} =$  parent's deadline
15:  end if
16:  remove  $I_i$  from  $G_l$ 
17: end while

```

Algorithm 2: Deadline relaxation during RMA

Relaxation proceeds by selecting the most critical task. If the subgraph G_l of the application graph G is circular, it is not immediately obvious which task is most critical because of circular producer-consumer relationships. Therefore, the algorithm selects the task in G_l with the shortest deadline. Because a critical task is eliminated from G_l at the end of each iteration, this selection has the effect of introducing cuts in G_l which removes circular paths. Otherwise, if G_l has no cycles, the most critical task in G_l is the one that doesn't consume data from other tasks in the subset. Before a task is removed from G_l , its deadline is relaxed by adding the deadline of its already removed parent, if the task *had* one. If the task had multiple parents, then the largest of its parents' deadlines is selected according to lines 8-12. This relaxation does not pose any limitation to the type of application graph that can be handled by the synthesis flow.

After relaxation, sorting begins. The first pass sorts according to task periods in ascending order. If the tasks do have differing periods, and G_l is at least partially connected, then a critical assumption is made that *if there is a node in G_l with a period less than that of any of its parent, then the edge between the node and the parent represents a weak precedence meaning that the corresponding task can execute without receiving data from its parent*. An example would be a task that infrequently obtains new parameters from another task for its internal computations. Otherwise, the application graph is faulty, and the resulting schedule is meaningless. Partial connectedness in this context means G_l is partially connected if G_l is not connected, and there exist at least one non-trivial subgraph of G_l that is connected.

The second pass sorts the table again according to deadlines, but the sorting is done only within rows containing the same period. Since deadlines have been previously relaxed, no distinction with respect

to precedence relationship between tasks needs to be made: if tasks I_{i_1} and I_{i_2} have no direct or indirect precedence, then I_{i_1} must finish before I_{i_2} if $D_{i_1} < D_{i_2}$, because I_{i_1} needs to finish earlier; if there is a precedence relationship, then $D_{i_1} < D_{i_2}$ because of the relaxation step, and I_{i_1} must appear before I_{i_2} . Indirect precedence in this context means there is a path from I_{i_1} to I_{i_2} via one or more intermediate tasks. Therefore, because second sorting is only done within rows with the same priority, tasks with shorter periods appear before those with longer periods regardless of whether or not latter tasks have shorter deadlines.

The sorting is topological, and is thus not unique. Moreover, if the group represents a non-connected graph, then the result after sorting is a partial order. The final order after sorting reflects the priorities in descending order, which are assigned by a simple enumeration.

With the table in place, the algorithm proceeds to compute the response time of each task in the group G_l according to the scheduling table. The response is computed recursively according to [89] as

$$r_i = T_{ij} + \sum_{\forall I_{i_h} \in G_l | \text{priority}(I_{i_h}) > \text{priority}(I_i)} \left\lceil \frac{r_i}{T_{i_h}} \right\rceil \cdot T_{i_h j} \quad (3.44)$$

This model of response time differs slightly from that of Liu and Layland [89] in that no bound in task blocking time due to safeguarding against priority inversion is included. This is because the programming model used here is message passing so that tasks do not share protected data so that semaphore-based synchronization for variables or memory locations is not required.

The analysis then concludes by comparing the response time against the execution time in lines 5-7 of Algorithm 1. The schedule feasibility parameter F_{lj} in (3.5) is set to zero if the response time is larger. Finally, the number of task switching is estimated in line (10) from the response time of the lowest priority task and from the period of the highest priority task. This estimate is conservative in that it is for the worst case by making the assumption that all tasks in the group are always ready when released so that the lowest priority task experiences maximum interference. The use of the parameter F_{lj} is explained in the following subsection.

The last parameter that is required before the ILP model data can be completely generated is the kernel/OS overhead O_j due to the clock interrupt handler. For cyclic executive, the overhead is zero. For fixed priority schedulers, the overhead can be estimated according to the analysis by Burns et al [91]. However, since the analysis is not yet implemented, the overhead was set to zero.

With all of the preceding parameters in place, the model data was generated as in the previous section for use with the solver lp_solve. Again, for both cases (cyclic and preemptive, WLAN and WCMDA), the same solver settings were used (node auto-ordering, most feasible basis crash, automatic branch & bound branching, and pre-solving of rows and columns).

The non-shaded rows in Table 3.15 summarizes the results, which were obtained on a machine with a T5500 processor and 2GB of memory. Therein, columns under ‘‘Cyclic’’ and ‘‘Priority-based Preemptive Scheduler’’ show the results under realtime and non-realtime constraints. The two columns under ‘‘problem size’’ show the number of constraints and decision variables of the ILP problem instance. The columns under ‘‘run time’’ show the time spent formulating and solving the problem respectively. The other two columns show the value of the objective function and the corresponding gap from the relaxed solution.

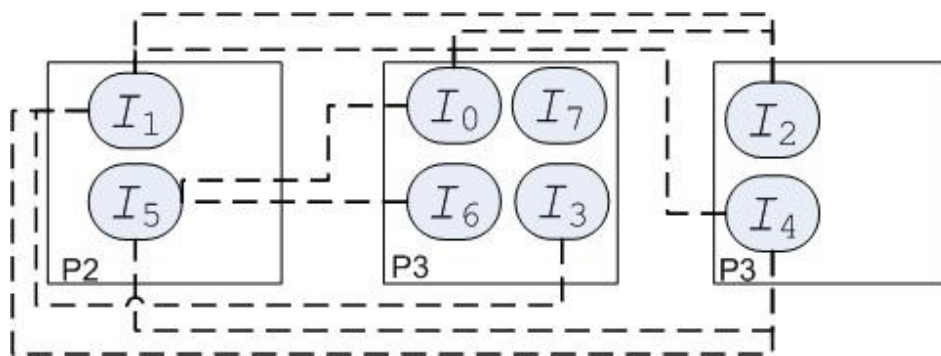
As expected, the additional time spent in formulation due to fixed priority scheduling is small compared to that spent in solver. The number of constraints and variables is still significant. The solver timed out in 3 out of 4 cases. Still, two of the 3 cases have gaps less than 18% so that synthesized architectures in those cases can be considered to be very good⁵. This result stresses again the importance of examining gaps as obtained from (3.34).

⁵18% is not a hard threshold through which one can state that solutions are good. A simple rule is that the smaller the gap the better.

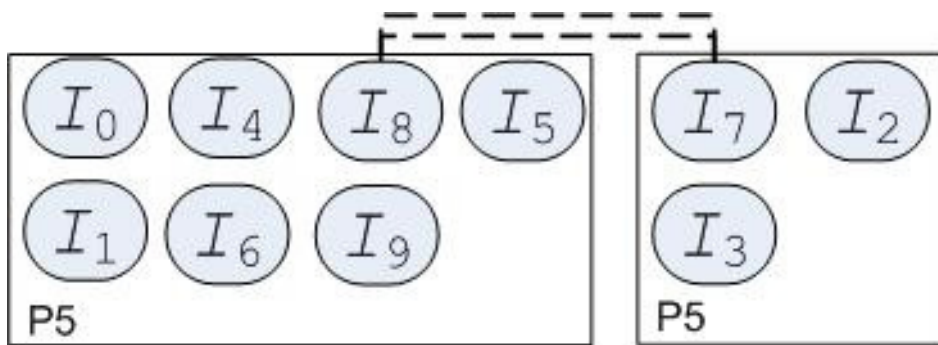
Figure 3.10 depicts the architectures. Also in this case the impact of expensive inter-task communication is evident: only a few processors are allocated. This cements the argument made in the previous section that examining traffic pattern versus task execution time is an effective tool for trimming the design space by reducing the number of processors without limiting the design space. For comparison, the shaded rows in the table shows the complexity when the number of processors instances is reduced to 2 for cores P1 to P2 each, and to 3 for cores P4 and P5 each, while the number of networks instances is 3 each for FSL and OPB. For WLAN, the solver run time was improved, and better values for the objective function were obtained.

Table 3.15: Synthesis results under scheduling extension (bold figures indicate suboptimum results; shaded rows are results for reduced number of processors and networks)

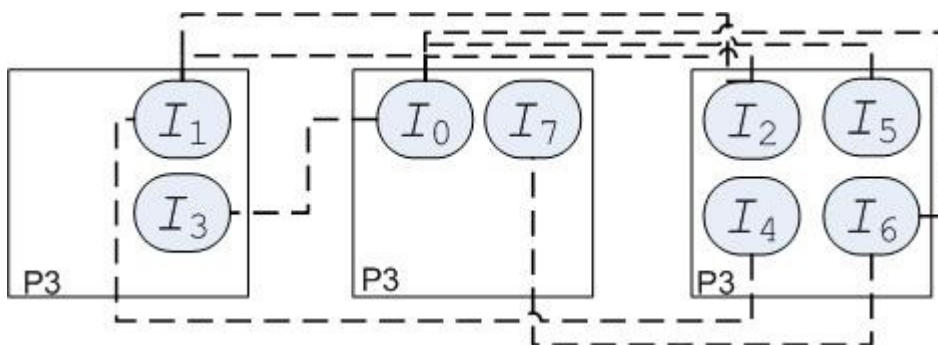
| Appl. | Cyclic | | | | Priority-based Preemptive Scheduler | | | | | |
|-------|--------------|----------------|-----------------|---------|-------------------------------------|----------------|-----------------|---------|--------------|--------|
| | problem size | run time (sec) | Objective (sec) | Gap (%) | problem size | run time (sec) | Objective (sec) | Gap (%) | | |
| WLAN | 29359 | 15023 | 497 | 28810 | 24661 | 10897 | 552 | 28811 | 18.44 | 1584.6 |
| WCDMA | 57312 | 28949 | 701 | 20654 | 43895 | 16354 | 704 | 28844 | 18.98 | 11.8 |
| WLAN | 7089 | 3638 | 497 | 635 | 5901 | 2538 | 552 | 1009 | 0.096 | 9.6 |
| WCDMA | 25394 | 12792 | 701 | 28810 | 25394 | 12792 | 704 | 19567 | 16.90 | 0.2 |



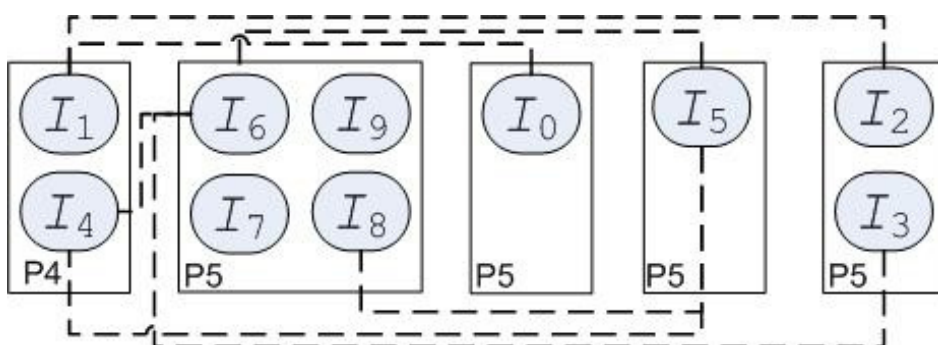
(a) WLAN with cyclic scheduler



(b) WCDMA with cyclic scheduler



(c) WLAN with priority-based scheduler



(d) WCDMA with priority-based scheduler

Figure 3.10: Synthesized architectures under scheduling extension (all networks are FSLs)

3.2.3 Makespan

This section presents a generalization of the model to optimize for the overall computation time or for the makespan of the parallel program.

Because the structure of targeted parallel programs as well as task deadlines, if any, are known a priori, tasks can be statically scheduled. The central problem in such situations is to schedule tasks such that the last finishing time of the computation to be performed (the makespan) is minimized. The determining factor during the process is the Critical Path (CP) of the Directed Acyclic Graph (DAG) representing the target parallel program. The CP in a DAG is the one with the largest sum of computation and communication costs. In contrast to the formulation up to this point, makespan minimization requires that the parallel program can be represented as a DAG, since otherwise a CP is not defined.

Scheduling algorithms which base on CP-mapping [92, 93] have been proposed for such problems. However, these cannot be applied to the automated synthesis problem for the following reasons:

1. Constituent components of the (heterogeneous) architecture are not known a priori since the makespan is minimized simultaneously with architecture determination. Consequently, for each node and edge in the DAG, there is generally more than one possible cost associated with feasible mapping and resource allocation. Consequently, the CP is generally not distinct.
2. Often, an optimum solution will require the sharing of processor resources between tasks, necessitating the use of a task-scheduler on processors, whose impact on the overall solution must be considered. This means that there is generally more than one possible mapping-dependent overhead cost that adds to task execution time, particularly when preemptive schedulers are used on processors.

To minimize the makespan, the objective function must capture the cost of the critical path only, which is not distinct. To solve that problem, we introduce a multi-pass combinatorial optimization technique which initially minimizes an arbitrary path. Algorithm 3 outlines the solution which has been implemented in the ILP formulator.

```

1: Fixed Paths  $\mathcal{F} = \{\}$ 
2: Path current,  $CP = \{\}$ 
3: Path last = longestPath( $G, \mathcal{J}, \mathcal{C}$ )
4: ILP solution sol =  $\{\}$ 
5: repeat
6:    $\mathcal{F} = \mathcal{F} \cup \textit{current}$ 
7:    $CP = \textit{last}$ 
8:   ILP instance ins = formulate( $CP, G, \mathcal{J}, \mathcal{C}, \mathcal{F}, \textit{sol}$ )
9:    $\textit{sol} = \text{solve}(\textit{ins})$ 
10:   $\textit{current} = \text{longestPath}(\textit{sol})$ 
11:   $\textit{last} = \textit{current}$ 
12: until delay(current) > delay( $CP$ )
13: return solution

```

Algorithm 3: Multi-pass optimization for makespan minimization

Starting with any path in a DAG, the intent is to capture the cost of the path in the objective function for minimization. With subsequent passes, the longest path is extracted from a previous pass, and is minimized in the current one. Synthesis stops when the longest path does not change. The rationale behind the method is that the total delay of the longest path after the last pass is the minimum makespan of the parallel program for given resources, *provided that the solution is optimum*. The

resulting task-mapping and architecture are thus the best one for the parallel program under the given resources.

In practise, a path may be the longest after a pass purely by chance because any feasible mapping for the path would suffice to fulfil the constraints in the pass *if the path is not in the objective function*. Therefore, it is possible that a multi-pass scheme would blindly jump to optimize one path after another, until an overall optimum architecture is found. This may lead to unpredictably long synthesis runs in the general case. It is thus desirable to somehow start with a good guess about the initial path to optimize, and quickly converge in subsequent passes.

For this purpose, the algorithm speculates by starting with the hypothetically longest path. This is the path with the longest delay, when for each node and edge in the DAG, the best possible resource is allocated out of the set of available processors \mathcal{P} and networks \mathcal{C} . A longest path problem is then solved after the initial allocation (Line 3). The CP obtained in this way is purely hypothetical because

1. All resource constraints are ignored in the allocation
2. It is assumed that each task in the DAG will exclusively occupy a processor, ignoring possible
 - i.) Scheduling overheads on processors
 - ii.) Reduction in path length due to processor sharing by eliminating network latencies

To converge to a solution without blindly jumping between paths, previous longest paths are fixed in subsequent passes. For this purpose, a set \mathcal{F} of fixed paths is used. The set, which is initially empty, specifies for ILP formulation the mapping of tasks on processors and the corresponding allocation of networks (Line 8). Therefore, fixing constrain the design space ahead of each pass. Since consequently sub-optimum architectures may result, this synthesis procedure essentially trades runtime against architecture optimality. Without fixing, optimality would be traded against uncertainty in runtime: the algorithm may never terminate in extreme cases.

The hypothetically longest path for the application graph is computed by way of a variation of the Dijkstra's algorithm for the shortest path problem [94]. The variation is that the algorithm starts by assigning an initial distance of negative infinity, and updates the distance and predecessor if, using the conventional notation for source and destination nodes u and v , $dist_between(u, v) + dist(u) > dist(v)$, rather than $dist_between(u, v) + dist(u) < dist(v)$.

Longest path problem is NP-complete. The heuristic above is intended to reduce the runtime for very large graphs, but has the property that under unfavorable weights in the graph it can return a path much shorter than the actual longest one, thereby leading to wrong decisions during makespan optimization.

During longest path computation in the first hypothetical pass, the task execution time T_{ij} on a best processor J_j is added to all incident edges of task I_i going to adjacent tasks in order to capture both the computation and the communication cost for longest path computation.

In contrast, the longest path after a pass is computed from the ILP solution by extraction and back annotation. The execution time T_{ij} is not that of the best processor, but that of allocated processor. Similarly, the communication cost results from allocated network. The computation cost is also added to incident edges going to adjacent tasks, but in this case, scheduling-related overhead is also added to the edges. Therefore, the longest path includes all actual time-related costs. The overhead is computed as shown below based on (3.35)

$$T_{switch_i} = \left(T_{s_{ij}} \cdot t_j + \frac{O_j}{|G_l|} \right) \quad (3.45)$$

where the number of switching $T_{s_{ij}}$ for a task $I_i \in G_l$ on a processor J_j is

$$T_{s_{ij}} = \begin{cases} \left\lceil \frac{r_i}{T_{i_h}} \right\rceil & \text{if there exist } I_{i_h} \in G_l \mid \text{priority}(I_i) = \text{priority}(I_{i_h})+1 \\ \left\lceil \frac{T_{ij}}{T_{clock_j}} \right\rceil & \text{for a round robin scheduler} \\ 0 & \text{otherwise} \end{cases} \quad (3.46)$$

and T_i is the period of a task in G_l . The switching overhead is differently computed depending on whether T_{ij} is defined for one period or for the whole duration of the program as discussed in Appendix B. The OS overhead is evenly distributed among all tasks in this model, but is zero for all experiments as mentioned in Section 3.2.2.

Three outcomes are possible with this algorithm. First, synthesis can terminate after a single pass with an optimum solution. Second, synthesis can terminate after several passes with a suboptimum solution due to fixing. This would occur if critical resources would be occupied by non-critical paths. Otherwise, synthesis would terminate with an optimum solution after several passes. For the latter two cases, it is not obvious after synthesis whether a solution is optimum. In such situations, repeating the synthesis with the last path as the initial CP will lead to the answer on optimality because the algorithm would terminate after one pass.

With the preceding discussion, the objective function for a terminating parallel program, or for one period of a non-terminating program, is the sum of the total execution time along the CP for the pass. The sum has three components, as given by

$$\min \left(\sum_{i=0}^n \sum_{j=0}^m x_{ij} \cdot T_{ij} \cdot X_i + T'_{net} + T'_{switch} \right) \quad (3.47)$$

This objective function has the same structure as (3.31). There are three differences. The first is that for each task I_i , there is a coefficient X_i whose value is 1 if the task is on the CP for a pass. This coefficient is specified in the ILP instance returned in Line 8 in Algorithm 3. For general application graphs when the objective is to minimize the total execution time of all tasks, each coefficient X_i is set to 1. The second difference is that an additional coefficient $Z_{i_1 i_2}$ is added in (3.30), leading to T'_{net} in the objective function.

$$T'_{net} = \sum_{I_{i_1}, I_{i_2} \mid I_{i_1} \prec I_{i_2}} \left(\sum_{k=0}^K \left((L_k D_{i_1 i_2} Z_{i_1 i_2} + \tau_k p_k B_{i_1, i_2}) z_{k i_1 i_2} + \sum_{j_1, j_2 \mid j_1 \neq j_2} \mathcal{L}_{k j_1 j_2} D_{i_1 i_2} \beta_{k i_1 i_2 j_1 j_2} \right) \right) \quad (3.48)$$

The coefficient is 1 if the corresponding edge in the DAG is on the CP for the pass. Only the latency term is weighted by the coefficient because if the edge is not on the CP then we do not wish to consider its cost, however mapping the edge to that specific resource would still lead to contention that may prolong data transfers for edges in a critical path.

Equation 3.48 is not accurate when an allocated network does not carry traffic for critical edges because the contention overhead within that resource would be included in the objective. A more accurate model would replace the variable $z_{k i_1 i_2}$ in the overhead with another 0-1 decision variable z'_k which becomes 1 when a critical edge is mapped to that resource:

$$z'_k \geq \frac{1}{E'} \sum_{I_{i_1}, I_{i_2} \mid I_{i_1} \prec I_{i_2}} z_{k i_1 i_2} Z_{i_1 i_2} \quad (3.49)$$

where E' is the number of edges in the CP. The accurate model for T'_{net} is thus

$$T'_{net} = \sum_{I_{i_1}, I_{i_2} | I_{i_1} \triangleleft I_{i_2}} \sum_{k=0}^K \left((L_k D_{i_1 i_2} Z_{i_1 i_2} z_{k i_1 i_2} + \tau_k p_k B_{i_1, i_2} z'_k) + \sum_{j_1, j_2 | j_1 \neq j_2} \mathcal{L}_{k j_1 j_2} D_{i_1 i_2} \beta_{k i_1 i_2 j_1 j_2} \right) \quad (3.50)$$

When the contention overhead is much smaller than actual latencies, (3.48) can be used in place of (3.49) and (3.50) to trade model size, and hence runtime, against accuracy.

The third difference is on the computation of context switching cost, where an extension to the RMA-based estimation Algorithm 1 is made. The extension is made because $T_{s_{l_j}}$ out of Line 10 in the algorithm cannot be directly used in the objective to minimize the makespan because a task in the CP cannot experience interference from lower priority tasks which are in the same group, but not on the CP. Since we only want to minimize interference for tasks on the CP, we cannot simply compute the response based on the lowest priority task in G_l . We need instead to compute the response based on the task with the lowest priority in the CP. This assumes that appropriate measures against priority inversion are in place. Therefore, in the proposed method, the switching cost is computed with a conditionally reduced number of tasks in the group to obtain $T'_{s_{l_j}}$. The condition is based on the priorities of non-critical tasks in the group.

Algorithm 4 shows how $T'_{s_{l_j}}$ is now computed. If the makespan is to be minimized, this reduced switching cost is used in (3.35), resulting to T'_{switch} in the objective. Otherwise, the complete responses of groups are used to minimize the total execution time of all tasks. Line 9 in the modified algorithm returns the priority of the task that is the lowest among the tasks in the CP based on the scheduling table. The priority in Line 10 is used to select tasks out of G_l that should be used to compute the largest response in line 11. Note that the selection includes all tasks from G_l which have higher priorities than the lowest priority task on the CP, even when a higher priority task is not in the CP. The reason is that, while such a higher priority task is not on the CP, its presence in the group contributes to switching overhead incurred by tasks with lower priority on the CP.

```

1: Table  $T = \text{createSchedulingTable}(G_l, J_j, \text{task time } T_{ij}, \text{task deadlines } D_i, \text{task periods } \bar{T}_i)$ 
2: for all  $I_i \in G_l$  do
3:   Response  $r_i = \text{computeResponse}(I_i, T)$ 
4:    $F_{l_j} = 1$ 
5:   if  $r > D_i$  then
6:      $F_{l_j} = 0$ 
7:   end if
8: end for
9: Priority  $p = \text{lowestPriorityTask}(CP, T)$ 
10: Select  $G'_l \subseteq G_l | \forall I_i \in G_l, \text{priority}(I_i) \geq p$ 
11:  $r = \text{computeLargestResponse}(G'_l, T)$ 
12:  $T'_{s_{l_j}} = \left\lceil \frac{r}{\text{period of highest priority task in } G_l} \right\rceil$ 

```

Algorithm 4: Determining context switching cost using RMA with makespan extension

The switching cost is thus given by

$$T'_{switch} = \sum_{l=0}^{2^n-2} \sum_{j=0}^m \gamma_{l_j} \cdot F_{l_j} \cdot (T'_{s_{l_j}} \cdot t_j + O_j) \cdot g_l \quad (3.51)$$

where g_l is 1 if the group G_l contains a task in the critical path, 0 otherwise. In non-makespan mode, g_l equals 1 for all groups.

Finally, since paths are fixed after each pass, the solution at the end of a pass is used to specify two further constraints that stipulate resource allocation for tasks and edges in the current longest path so that these are excluded from optimizations in subsequent passes. The values of parameters ϵ_{ij} and ε_{ij} in constraints (3.52) and (3.53) are thus set according to the solution depending on whether or not the node and the edge is on the critical path. The value is 1 if the corresponding node or edge has a fixed allocation for that particular resource. The allocation of processors and networks is annotated in nodes and edges of the application graph so that the fixed allocation can be used for ILP formulations during subsequent passes. In non-makespan mode, all of these two type of parameters carry the value 0.

$$x_{ij} \geq \epsilon_{ij} \quad \forall I_i \in \mathcal{I}, J_j \in \mathcal{J} \quad (3.52)$$

$$z_{ki_1i_2} \geq \varepsilon_{ij} \quad \forall C \text{ and } I_{i_1}, I_{i_2} \in \mathcal{I} \mid I_{i_1} \prec I_{i_2} \quad (3.53)$$

3.2.3.1 Experimental Results

This section presents experimental results aimed at studying the feasibility of ILP-based synthesis after extending the model to include makespan minimization. The WLAN application from Section 3.2.2 is used with same processors and networks, with the same reduced number of instances. WCDMA and the six other applications cannot be used because their implementations are such that they are not presentable as DAGs.

Also in this case, two runs with and without realtime scheduling were conducted. The same synthesis machine was used. Figure 3.11 shows the DAG representing the application graph to assist with understanding results from various paths.

Hypothetical path lengths were computed based on Tables 3.12, 3.10 and 3.5. The results are given in the 3rd column of Table 3.16, which shows that p1 is the hypothetically longest one. Each row in the table shows the length of a particular path in the application DAG when certain paths are optimized.

Even though the makespan was much less without real time scheduling in each run as compared to optimizing under preemptive scheduling, the resulting architectures could not meet real time requirements for the application. This is because, without deadline guarantees for any of the tasks, the implementation must be fast enough to complete all processing before the deadline of any task. However, from Table 3.10, we note that no deadline can be guaranteed. Under preemptive scheduling, all deadlines are guaranteed because of scheduling constraints. If it is desired to just meet the deadlines and minimize the system area instead, the roles of (3.27) and (3.47) can be swapped during optimizations.

With p1 as the initial pass, the synthesis terminated with optimum results after one pass in each case. The minimum makespan is significantly shorter in each case (columns p1) compared to hypothetical lengths in the third column because of processor sharing. This is expected because inter-task communications are expensive in this parallel program so that optimizing for p1 tends to minimize inter-processor communications.

To compare results with and without makespan minimization, synthesis runs were conducted in which all paths were included in the objective (columns “all”). Because path p1 is dominant, the same results were obtained as with makespan minimization with p1. However, we expect different results in general, because optimizing for all paths could proceed at the expense of the CP.

Finally, to demonstrate what happens with multiple synthesis passes, two further sets of synthesis runs were conducted with p0 and p8 as initial paths (rather than the hypothetically longest p1). Starting with p0, non-realtime synthesis terminated with optimum results after one pass (length of p1 under p0 = length of p1 under p1 = 159 ns). For the realtime case, the synthesis also terminated after one

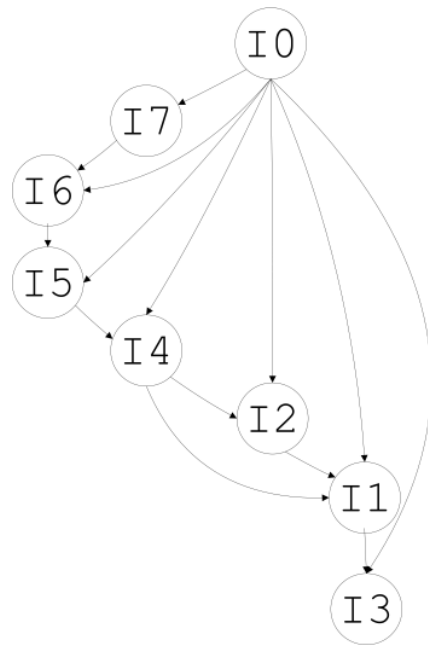


Figure 3.11: WLAN Program Graph

pass, but with suboptimum results (length of p1 under p0 (95294 ns) > length of p1 under p1 (93031 ns)).

Starting with p8, the first pass ended with suboptimum results, indicating that p1 needs to be optimized instead. In the second pass, p8 was fixed, and p1 was optimized (column p1*). For the non-realtime case, the second pass again ended with optimum results (length of p1 under p1* = length of p1 under p1 = 159 ns). For the other case, p0 became the longest path (94911 ns) after the second pass. Fixing p8 and p1 while optimizing p0 in the third pass did not improve the results. These results demonstrates that starting with paths other than the hypothetically longest one can lead to suboptimum results as suggested earlier.

To conclude this section, Table 3.17 shows problem sizes, run times and gaps for each of the above synthesis pass.

3.3 Chapter Summary

This chapter introduced an ILP model to enable an automatic synthesis of flexible CMP systems where emphasis was placed in inclusiveness of problem dimensions and joint optimization of design subspaces so as to guarantee globally optimum solutions.

Since this model is not pre-constraining, multiple communication resources can be allocated between any pair of processors as observed in experimental results. Such allocation schemes minimize expensive inter-task communications. Thus, the generated architecture description includes not only the netlist, but also information on which communication libraries a task should use to communicate with any other task. This is different from other proposed synthesis and exploration methodologies in related work where there are no multiple allocations, and all tasks on a given processor are often required to share networks a priori.

In all experiments, the message passing paradigm was used. The assumption for the physical implementation of messaging passing interfaces is that FIFOs are used to queue messages. This means that, once a task has initiated a data transfer by calling the appropriate function, the task is free to

Table 3.16: Makespan path lengths for WLAN

| Idx. | Path | Hy. len (μ s) | Length (μ s) for a non-real time kernel when optimizing path | | | | | Length (μ s) for a real time kernel when optimizing path | | | | |
|------|-------------------------|--------------------|--|------------|------------|-----------------|------------|--|--------------|-----------------|--------------|-----|
| | | | all | p1 | p0 | p8 | p1* | all | p1 | p0 | p8 | p1* |
| p0 | I0,I7,I6,I5,I4,I1,I3 | 32993230 | 155 | 155 | 154 | 42189263 | 155 | 93027 | 85070 | 42135205 | 94911 | |
| p1 | I0,I7,I6,I5,I4,I2,I1,I3 | 32993234 | 159 | 159 | 159 | 42189268 | 159 | 93031 | 95294 | 42135209 | 84695 | |
| p2 | I0,I6,I5,I4,I1,I3 | 162550 | 126 | 126 | 126 | 161487 | 126 | 92998 | 85041 | 107462 | 94882 | |
| p3 | I0,I6,I5,I4,I2,I1,I3 | 162555 | 130 | 130 | 130 | 161491 | 130 | 93002 | 95265 | 107466 | 84666 | |
| p4 | I0,I5,I4,I1,I3 | 162526 | 101 | 101 | 101 | 161488 | 101 | 93006 | 85050 | 107437 | 94891 | |
| p5 | I0,I5,I4,I2,I1,I3 | 162530 | 105 | 105 | 105 | 161491 | 105 | 93010 | 95273 | 107442 | 84675 | |
| p6 | I0,I4,I1,I3 | 162512 | 87 | 87 | 87 | 161481 | 87 | 92992 | 85035 | 107423 | 94876 | |
| p7 | I0,I4,I2,I1,I3 | 162516 | 91 | 91 | 91 | 161485 | 91 | 92977 | 95259 | 107427 | 84660 | |
| p8 | I0,I2,I1,I3 | 37863 | 88 | 88 | 88 | 86 | 88 | 5156 | 5593 | 4716 | 4716 | |
| p9 | I0,I1,I3 | 37859 | 84 | 84 | 84 | 81 | 84 | 4712 | 5589 | 4712 | 5152 | |
| p10 | I0,I3 | 93 | 68 | 68 | 68 | 65 | 68 | 3354 | 3782 | 3346 | 3354 | |

* while fixing p8

Table 3.17: Synthesis results for makespan minimization for WLAN (bold figures indicate possibly suboptimum results)

| Cyclic | | | | | | | | | | Priority-based Preemptive Scheduler | | | | |
|--------|---------|--------|-------|--------|---------|--------|-------|--------|--------------|-------------------------------------|-----------------|---------|--|--|
| Path | # Cons. | # Var. | Form. | Solver | # Cons. | # Var. | Form. | Solver | Problem size | Run time (sec) | Objective (sec) | Gap (%) | | |
| all | 7089 | 3638 | 497 | 635 | 0.0002 | 5901 | 2538 | 552 | 1009 | 0.096 | 9.6 | | | |
| p1 | 7089 | 3638 | 498 | 2042 | 0.0002 | 5901 | 2538 | 500 | 1091 | 0.0944 | 9.4 | | | |
| p0 | 7089 | 3638 | 501 | 2524 | 0.0002 | 5901 | 2538 | 4389 | 4388 | 0.085 | 8.5 | | | |
| p8 | 7089 | 3638 | 483 | 28799 | 0.0001 | 5901 | 2538 | 491 | 28801 | 0.005 | 0.4 | | | |
| p1* | 7089 | 3638 | 501 | 2526 | 0.0002 | 5901 | 2538 | 4386 | 4388 | 0.085 | 8.5 | | | |

do further processing and to initiate or wait for data via another communication resource. Obviously, communication latencies can be hidden through overlapping. However, this effect is not accounted for in the objective function because temporal information is not used. Consequently, the actual cost in total computation time can be smaller than what the value of the objective function after synthesis would indicate.

Accounting for temporal information is not a feasible prospect because a far greater number of variables would need to be considered to model and capture all possible moments in which communications can be initiated. This aspect is not unique for automated synthesis: workbench based methodologies require simulations, but these simply represent a specific scenario.

Evaluation results of ILP-based synthesis lead to the following conclusions:

Solver runtime is a challenge. The number of tasks and that of processors in an ILP instance significantly influence the runtime. It is thus important that alternative synthesis approaches based on this ILP model are investigated.

ILP complexity can be reduced without limiting the design space. This can be achieved by reducing the number of processors by comparing traffic pattern and task execution time so that the number of processors can be potentially reduced. Nevertheless, when problem sizes become huge as one would expect for future systems, the effectiveness of this technique will be limited.

Useful results can be obtained even when the solver times out. A designer should always examine gaps between relaxed and integer solutions when an ILP-based synthesis is conducted to get a measure on the quality of results obtained.

Automated synthesis is absolutely important. The combinatorial nature of the problem can lead to counter intuitive results. Without an automated approach, even a skilled designer can easily miss superior design points.

The approach presented herein enables a wide and disciplined coverage of the design space provided each synthesis run during exploration is fast. Next chapter presents an alternate synthesis method based on SAT techniques to address runtime problems. The ILP model presented in this chapter is the foundation for the work in next chapters.

4 SAT-Based Synthesis

Evaluation results for ILP-based synthesis in the previous chapter showed that solver runtime is very long for large problem instances. Thus, motivated by recent advances in resolving Boolean Satisfiability (SAT) problems [95, 96, 97], this chapter discusses how well methods used in the SAT domain can be exploited to solve the synthesis problem.

A SAT problem is a decision problem of determining whether variables of a given boolean formula can be assigned such that the formula evaluates to logical value *true*, or whether such an assignment does not exist. Section 4.1 presents a summary of techniques used in modern SAT solvers.

0-1 ILP problems can be solved using SAT techniques, and even using existing SAT tools, provided the problems are appropriately translated. Since translation leads to a huge number of literals and SAT-clauses [98], we opted to search for reportedly good solvers with native support for 0-1 ILP problems. One such solver is clasp [95] from the domain of Answer Set Programming (ASP) which is closely related to the SAT domain. This solver is a part of a suite called Potassco for Answer Set Programming. While clasp is strictly not a SAT solver, it uses standard SAT techniques for ASP solving. Therefore, the discussion in section 4.1 is equally relevant to clasp.

Following the discussion on modern SAT techniques, section 4.2 briefly introduces ASP and Potassco tools, and describes how the synthesis problem formulated in the previous chapter is encoded using ASP semantics. Finally, section 4.4 presents a comparison of synthesis results obtained using ASP versus results obtained in the foregoing chapter. Concluding remarks in section 4.5 provide a motivation for greedy-like synthesis heuristics which were developed in this research to address runtime problems for large problem instances.

4.1 Methods for Solving SAT Problems

Given a boolean formula, a SAT solver determines whether there exist an assignment of *true/false* values to variables in the formula, for which the formula evaluates to *true*. The formula is usually represented in a Conjunctive Normal Form (CNF). In the form, a formula φ is represented as a conjunction of clauses c_i , each of which is a disjunction of literals p_i . Literals denote either a variable x or its complement \bar{x} .

4.1.1 The Resolution Proof System and the DP Algorithm

The unsatisfiability of a formula can be proven through a Resolution Proof System (RPS) introduced by Robinson in 1965 [99]. RPS is based on the resolution operation. If c_1 and c_2 are two clauses, the resolution operation creates a clause c , the resolvent, such that $c_1 \wedge c_2 \rightarrow c$, meaning that if c_1 and c_2 are satisfied, then c is also satisfied. The resolution operation works as follows:

1. If c_1 and c_2 have a pair of opposite literals p and $\neg p$, then $c = \{c_1 \cup c_2\} \setminus \{p, \neg p\}$, i.e. the resolvent contains all literals from the two clauses without the opposing pair of literals.
2. If c_1 and c_2 do not have a pair of opposite literals, then $c = \{c_1 \cup c_2\}$.

It is intuitive why $c_1 \wedge c_2 \rightarrow c$: noting that a clause is satisfied when at least one literal in it has a *true* assignment, it is obvious that a satisfiable assignment of c is also a (partial) satisfiable assignment of c_1 and c_2 .

RPS proceeds by applying the resolution operation to all possible pairs of clauses containing opposing literals. The resolvents are added by conjunction to the boolean formula if they are not tautologies, and resolution is applied to the new formula. If an empty clause is derived, then the original formula is unsatisfiable, because, by definition, an empty clause cannot be satisfied. If the formula is not satisfiable, then the RPS procedure always terminates. However the procedure does not terminate for the general SAT problem because satisfiability is not a decidable problem for predicate logic [99].

The Davis Putnam (DP) algorithm that was introduced in 1960 [100], earlier than Robinson's work, is based on resolution. The algorithm works as follows:

1. Choose a literal p out of φ
2. Find all resolvents of φ on p
3. Remove from φ all clauses of φ that contain the literal p , add the resolvents to φ and simplify
4. If φ becomes an empty clause, then the original formula is not satisfied, else if no clauses are left, the original formula is satisfied, else the procedure is repeated on the new formula.

Because in step 3 clauses are removed and new ones are added, the resulting formula in each iteration is not equivalent to the one before it. Still, satisfiability is preserved because the added resolvents are satisfied if and only if removed clauses are satisfied.

The major weakness of the DP algorithm results from step 3 because the number of added clauses can grow exponentially. A refinement of the algorithm was introduced in 1962, and is commonly known as the Davis-Putnam-Logemann-Loveland (DPLL) algorithm.

4.1.2 The DPLL Algorithm

In contrast to the DP algorithm, the DPLL algorithm searches for truth assignments while performing propagation of unit clauses and backtracking. Unit clauses are those which contain only a single unassigned literal, and thus can always be satisfied by assigning an appropriate value to the literal. During propagation, all clauses that contain the literal are eliminated because they can be satisfied by the assignment. All clauses that contain the opposing literal are shortened by removing the literal. As with the DP algorithm, this procedure preserves satisfiability even though the resulting formula is not equivalent. Similarly, if a formula becomes empty, then the original formula is satisfied, or if an empty clause is derived, then the original formula is not satisfied.

The backtracking part is performed by assigning a value to a literal, and proceeding with unit propagation. If an empty clause is derived, the literal is assigned the complementary value followed by unit propagation as follows:

1. Perform unit propagation for all unit clauses.
2. If φ is empty, then the formula is satisfied, else
3. If an empty clause is derived, then the formula is not satisfied, else
 - i.) Select a literal p out of φ
 - ii.) Go back to step 1 with $p = 1$, if not satisfied
 - iii.) Go back to step 1 with $p = 0$

Modern efficient SAT solvers are based on the DPLL algorithm. Significant improvements have been achieved through a technique called clause learning, and through efficient unit propagation. These two improvements are the base of state of the art solvers, including the ASP solver clasp.

4.1.2.1 Clause Learning

Clause learning is a technique that improves the DPLL algorithm by analyzing reasons leading to a conflict, and exploiting the knowledge learned. A conflict occurs in the DPLL algorithm when a variable needs to be assigned opposing truth values during unit propagation. Conflicts can be illustrated through a decision graph. Figure 4.1 is a decision graph for an example formula

$$\varphi = (p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3 \vee p_4) \wedge (p_2 \vee \neg p_3) \wedge (\neg p_2 \vee \neg p_3 \vee \neg p_4)$$

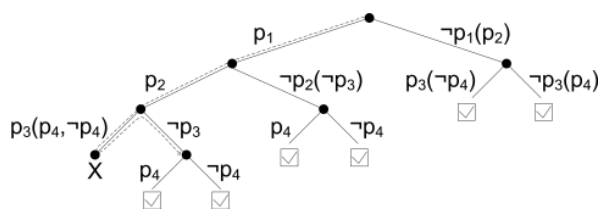


Figure 4.1: A decision graph for the DPLL algorithm

A node in the decision graph represents the assignment of a truth value to a literal in an iteration of the DPLL algorithm, and is thus a decision point. A literal that is assigned a value in this manner is called a decision literal. The truth value assigned to a decision literal is indicated on the edges, so that an edge represents a path from one iteration to another in the algorithm. Unit propagation is indicated by brackets next to decision literals. If the assignment of a value to a decision variable followed by unit propagation leads to the implication that a certain literal must be assigned a particular truth value to satisfy a given clause, then that assignment is indicated in the brackets. For example, assigning the value *false* to the decision literal p_1 implies the assignment *true* to the literal p_2 in order to satisfy the first clause in φ . Literals which are assigned values following unit propagations are called implied literals.

Unit propagations can lead to conflicts in the assignment of implied literals. For instance, assigning the value *true* to literals p_1 and p_2 do not lead to implied assignments. This partial assignment satisfies the first and the third clause. However, further assigning the value *true* to p_3 implies the values *true* and *false* for p_4 in order to satisfy the second and the fourth clause respectively. Such a conflict is indicated by X in the figure, and the cause is called a conflict literal. Thus the DPLL algorithm will have to backtrack to the last decision point, and assign the opposite value to the decision literal. A tick in the figure indicates a complete satisfying assignment. The dotted line marks a path that the algorithm could take in a run. Contrary to this illustration, a decision graph usually indicates the actual path that the algorithm has taken.

The decision graph reflects the iterative nature of the DPLL algorithm, and thus contains the information on assignment decisions made. However, information on partial assignments that lead to a particular conflict is not readily available, therefore, a complimentary data structure called an implication graph is used for that purpose. Figure 4.2 shows the implication graph for the foregoing example.

A node in an implication graph represents an assignment decision for the corresponding literal. Each decision is associated with a level which is indicated in a bracket next to the literal. Edges between nodes signify assignment implications due to unit propagation. In the example, the implication graph is constructed as follows: p_1 and p_2 are assigned in the first two decision levels, so corresponding

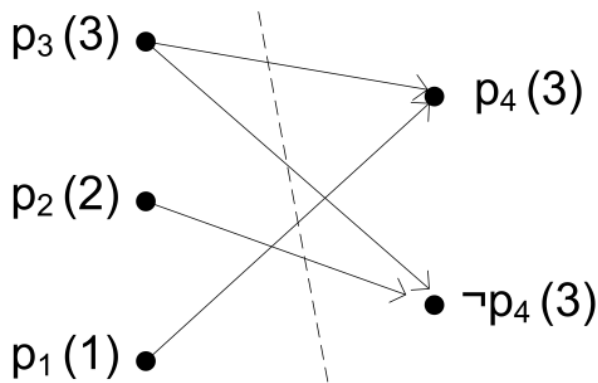


Figure 4.2: Implication graph for the foregoing example

nodes are created. p_3 is then assigned in level three. To satisfy the second clause, p_4 is assigned true in level three. This assignment is implied by decision for p_1 in level 1, and p_3 in level three, therefore, edges are created to the node for p_4 . Similarly, edges are created to the node for $\neg p_4$.

The procedure for determining a partial assignment for a conflict based on an implication graph proceeds by partitioning the graph in two [101]. The partition has all decision literals on one side, called the decision side, and the conflict literal on the other, called the conflict side. All vertices on the decision side that have at least one edge crossing over to the conflict side constitute a partial assignment leading to the conflict, whose disjunction is called the conflict clause. This process is known as conflict analysis, and adding clauses to the formula is called learning [101].

Satisfying a conflict clause has the effect of preventing the conflict assignment from occurring, thus effectively pruning the search space. This is equivalent to backtracking more than one level in the decision graph in contrast to the original DPLL algorithm, therefore, the technique is called non-chronological backtracking [101].

4.1.2.2 Fast Unit Propagation

Significant improvements can be achieved by employing efficient mechanisms for unit clause propagation because the operation is used extensively in DPLL-based SAT solvers [102]. An efficient mechanism was first introduced for the SATO [102] solver.

A key aspect to the mechanism is a data structure that reduces the work that needs to be performed during unit clause propagation. During propagation, when a variable is assigned a value, there is a need to determine whether a clause that contains a literal of the variable becomes satisfied, unsatisfied, or unit. Consequently, data structures are implemented such that there are references from variables to clauses containing their literals. With a large number of variables and clauses, traversing such references is an expensive operation. The method introduced with the SATO solver speeds up this operation by keeping a reduced number of references.

The reduction makes use of the fact that not all clauses which contain a literal of a variable require updating when the variable is assigned a value. To conclude whether a clause has become satisfied, unsatisfied, or unit, one only needs to examine the last unassigned literal. Thus, in order to efficiently determine the assignment of the last literal, SATO maintains two references to clauses, called the head and the tail, which initially point to the first and last literal in the clause so that a clause is referenced to from two variables only. The order of literals in clauses is arbitrary.

Whenever a variable that references to the head or tail is assigned a *false* value, the head/tail is moved in the list in search of the next unassigned literal. In case an unassigned literal is reached, then a new reference is created from its corresponding variable to the literal, and the new reference becomes the

head/tail. If a satisfied literal is found, then the clause becomes satisfied. If the head/tail reaches the other reference, then they point to the only unassigned literal so that the clause becomes unit. The advantage of this data structure is that the status of a clause is updated only when the head/tail is assigned a *false* value, eliminating unnecessary updates when other literals in it are updated.

SATO maintains a sense of order between the two references: all literals before the head and all literals after the tail must be assigned the value *false*. During a search, one only needs to move the two references towards each other. When the search process backtracks, the previous head and tail references needs to be restored, implying additional work in the process. The data structure used in the Chaff solver [103] trades efficiency in searching for unassigned clauses for backtracking efficiency. That is achieved by eliminating the sense of order between the two references. As a consequence, all literals needs to be traversed in search of the next unassigned literal. However, since no order needs to be preserved during backtracking, the two references can remain intact in the process.

4.2 Answer Set Programming and the Potassco Suite

Answer Set Programming (ASP) is form of declarative logic programming oriented towards difficult search problems. ASP is different from procedural programming in that a problem is described using a formal language, and a solver finds a solution. A problem is presented as a logic program which consists of a set of atoms and rules [104].

An atom is a Boolean proposition about the problem universe, whereas rules specify relationships between the atoms. A solution to a program is called a stable model and tells which atoms are true [104]. This is similar to SAT problems if rules and stable models are perceived to be clauses and satisfying assignments respectively.

Besides atoms and rules, logic programming languages contain two further elements: constants and variables. The former are either numeric or symbols, whereas the latter are strings used for generalization. When writing a logic program, variables are not assigned values, but rather, the solver finds values for them. Atoms consist of symbols followed by a parenthesized list of constants or variables: for instance, in *processor(risc, ppc)*, *risc* and *ppc* are constants, and the atom *processor* might tell us if *ppc* is a *risc* architecture provided we have specified rules from which the solver can derive the truth value of the atom. Rules consist of a head and tail separated by the connective “:-” which corresponds to a reversed implication “ \leftarrow ”, i.e., the atom in the head is derived (to be true) if all comma-separated atoms in the tail are derived.

ASP problems are typically solved by first removing variables from programs through substitutions, and then passing the resulting equivalent variable-free program to the solver. The former part is called grounding, and is either done using a separate tool or in an integrated grounder-solver. Grounding is required because current ASP solvers work on variable-free programs only.

Potassco, the Potsdam Answer Set Solving Collection, is a suite of tools developed at the University of Potsdam which consists of the solver clasp, a grounder Gringo, a combination of clasp and Gringo called Clingo, and an incremental version of Clingo called iClingo [105]. The solver clasp uses, among others, conflict-driven learning and unit propagation. We opted to use this tool to study how well SAT-based techniques can be exploited for this synthesis problem because benchmark results have shown that learning solvers do perform well at least for structured problems [106], as is the case with synthesis problems at hand. The next section discusses how the automated synthesis problem as modeled in Chapter 3 is specified using the language supported by Gringo.

4.3 Encoding the Synthesis Problem as ASP Programs

The language natively supported by Gringo can express linear constraints for 0-1 ILP problems. Linear inequalities are coded into rules whose general form is

$$b [v_0 = a_0, v_1 = a_1, \dots, v_n = a_n] c. \quad (4.1)$$

where the syntax $v_i = a_i$, $a_i \in \mathbb{N}$ denotes the weight a_i of a variable v_i in a linear (in)equality, and the syntax $b [\dots] c$, $b, c \in \mathbb{N}$ is a general form for constraining such that a coded constraint represents an equality if $b = c$, a less-than inequality if b is not specified (is absent), and a greater than inequality if c is not specified [105]. Since weights in the ILP model are generally out of \mathbb{R} whereas $a_i, b, c \in \mathbb{N}$, rounding is required. We therefore round a_i and b up, and c is rounded down. Consequently, more restrictive constraints result which can theoretically exclude a solution that otherwise does not violate constraints of the original problem instance.

The rest of this section uses the same notation for variables and parameters as in Chapter 3.

4.3.1 Task Mapping

Task mapping constraint (3.1) is coded for each I_i as

$$1 [x_{i0} = 1, x_{i1} = 1, \dots, x_{im} = 1] 1. \quad (4.2)$$

This is a ‘‘constrained choice rule’’ which says: choose at least one, but at most one, of the atoms $x_{i0}, x_{i1}, \dots, x_{im}$, so that each task is mapped to one processor only.

4.3.2 Processor Sharing

The processor sharing constraint (3.3) is coded by setting the upper bound c to the size of the program memory s_j

$$[x_{0j} = s_{0j}, x_{1j} = s_{1j}, \dots, x_{nj} = s_{nj}] s_j. \quad (4.3)$$

The sizes s_{ij} of the tasks in program memories of processors are rounded up, whereas the sizes s_j are rounded down. The omission b in (4.3) reflects the fact that we do not wish to specify a lower bound so that a processor J_j may remain unallocated. This rule insures that mappings do not exceed the capacity of any processor.

4.3.3 Processor Area Constraints

The area constraint for processors on the FPGA is specified by three constraints. The first (3.10) which insures that a variable v_j for a virtual processor is greater than one when at least one task is mapped on a processor J_j is coded as

$$[x_{0j} = 1, x_{1j} = 1, \dots, x_{nj} = 1, v_j = -(n + 1)] 0. \quad (4.4)$$

The second constraint (3.11) which forces v_j to zero when J_j is not allocated is coded as

$$[x_{0j} = -1, x_{1j} = -1, \dots, x_{nj} = -1, v_j = 1] 0. \quad (4.5)$$

The last constraint (3.12) involves non-integral processor area (coefficients a_j) and FPGA area for processing elements A_{PE} . These are rounded up and down respectively in the following coding

$$[v_0 = a_0, v_1 = a_1, \dots, v_m = a_m] A_{PE}. \quad (4.6)$$

4.3.4 Network Usage Constraints

A number of linear constraints were introduced in Section 3.2.1.4 in order to decompose constraint (3.13) into a series of simpler linear constraints. The purpose of the constraint is to insure that if two communicating tasks are mapped on different processors, then a network resource is allocated for the two tasks.

The introduction of the auxiliary variable $\alpha_{i_1 i_2 j_1 j_2}$ was the first step in the decomposition process. We therefore wish to encode resulting constraints in ASP semantics. But since (3.14) is already a conjunction of variables $x_{i,j}$, we can encode (3.14) directly as a rule so that (3.15) and (3.16) can be dropped to reduce the number of constraints:

$$\alpha_{i_1 i_2 j_1 j_2} \leftarrow x_{i_1 j_1}, x_{i_2 j_2}. \quad (4.7)$$

The next constraint (3.18) in the decomposition process is coded as

$$0 [\lambda_{i_1 i_2} = -1, \alpha_{i_1 i_2 j_0 j_1} = 1, \alpha_{i_1 i_2 j_0 j_2} = 1, \dots, \alpha_{i_1 i_2 j_m j_{m-2}} = 1, \alpha_{i_1 i_2 j_m j_{m-1}} = 1] 0. \quad (4.8)$$

so that $\lambda_{i_1 i_2}$ is derived when two corresponding communicating tasks are mapped on different processors. Similarly, constraints (3.22), (3.23) and (3.24) are coded as

$$0 [\lambda_{i_1 i_2} = -1, z_{k i_1 i_2} = 1, z_{k i_1 i_3} = 1, \dots, z_{k i_n i_{n-2}} = 1, z_{k i_n i_{n-1}} = 1] 0. \quad (4.9)$$

$$[y_k = -1, z_{k i_1 i_2} = 1] 0. \quad (4.10)$$

$$[\alpha_{i_1 i_2 j_1 j_2} = -1, z_{k i_1 i_2} = 1] V_{j_1 j_2 k}. \quad (4.11)$$

4.3.5 Network Capacity and Area Constraints

Constraints for network capacity (3.25) and FPGA area for the communication infrastructure (3.26) are coded as

$$[y_k = 1, z_{0 i_1 i_2} = 1, z_{1 i_1 i_2} = 1, \dots, z_{k i_1 i_2} = 1] M_k. \quad (4.12)$$

$$[y_0 = A_0, y_1 = A_1, \dots, y_k = A_k] A_{net}. \quad (4.13)$$

The parameters M_k are integers and require no further consideration. However, as in previous subsections, the coefficients A_k and the parameter A_{net} are rounded up and down respectively so that (4.13) is more constraining compared to (3.26).

The auxiliary variable $\beta_{k i_1 i_2 j_1 j_2}$ that captures latencies through network bridges is essentially a conjunction of variables $\alpha_{i_1 i_2 j_1 j_2}$ and $z_{k i_1 i_2}$ so that (3.28) and (3.29) can be coded using one rule

$$\beta_{k i_1 i_2 j_1 j_2} \leftarrow \alpha_{i_1 i_2 j_1 j_2}, z_{k i_1 i_2}. \quad (4.14)$$

4.3.6 Scheduling Constraints

The scheduling feasibility constraint for a group of tasks on a particular processor (3.36) is coded as

$$[\mathcal{M}_{l_j} = 1] F_{l_j} \quad (4.15)$$

The link between the auxiliary variable \mathcal{M}_{l_j} for a group of tasks and mapping decision variables x_{ij} (3.37) is already a conjunction of variables so that the constraint can be specified directly as a rule

$$\mathcal{M}_{l_j} \leftarrow x_{i_{l_0} j}, x_{i_{l_1} j}, \dots, x_{i_{l_g} j} \quad (4.16)$$

thereby dropping (3.39) and (3.40) to reduce the problem size. However, (4.16) represents a logical implication, whereas (3.37) is a logical equality. Without further measures, a stable model can potentially have \mathcal{M}_{l_j} as true when one or several of the atoms $x_{i_{g_j}}$ are false. We therefore additionally add the rule

$$\leftarrow \mathcal{M}_{l_j}, 1[\text{not } x_{i_{0j}}, \text{not } x_{i_{1j}}, \dots, \text{not } x_{i_{g_j}}]. \quad (4.17)$$

as an integrity constraint [105] such that \mathcal{M}_{l_j} is not derived if any of associated atoms is not derived. Constraints (3.42) and (3.43) which together set the value of the auxiliary variable γ_{l_j} such that scheduling cost is captured for the largest group on a processor only cannot be directly translated to equivalent ASP facts since rounding up the weights $1/|I|$ to meet integrality requirement destroys the property of the two constraints. However, a closer look reveals that the two constraints essentially stand for logical conjunctions which can be represented by the rule

$$\gamma_{l_j} \leftarrow \mathcal{M}_{l_j}, \text{not } \mathcal{M}_{l_{s_0j}}, \text{not } \mathcal{M}_{l_{s_1j}}, \dots \quad (4.18)$$

where $\mathcal{M}_{l_{s_ij}}$ is the i th super set of a group G_l . The implication is that γ_{l_j} is derived when the corresponding group \mathcal{M}_{l_j} is derived, but none of the atoms for associated super groups are derived.

4.3.7 Makespan

For makespan mode, we need to specify rules which govern whether or not a node or an edge in the application graph is on the critical path. This is done in the objective function in the following subsection. Additionally, we need to specify if a node or an edge is fixed in any given pass of Algorithm 3, therefore, the following two rules are added based on (3.52) and (3.53) respectively:

$$\epsilon_{l_j} [x_{ij} = 1]. \quad (4.19)$$

$$\epsilon_{ij} [z_{ki_1i_2} = 1]. \quad (4.20)$$

Here, the less expensive model (3.48) is used so that (3.49) remains unspecified.

4.3.8 Objective Function

The objective function (3.47) is coded using an optimization statement by way of the “minimize[]” syntax [105]:

$$\text{minimize} [\dots, \gamma_{l_j} = (T'_{s_{l_j}} \cdot t_j + O_j), \dots, x_{ij} = T_{ij}, \dots, z_{ki_1i_2} = (L_k D_{i_1i_2} + \tau_k p_k B_{i_1, i_2})]. \quad (4.21)$$

The weights T_{ij} for task execution times are set to zero when optimizing in makespan mode if the corresponding task is not on a critical path. Similarly, weights for the sum of scheduling $T'_{s_{l_j}} \cdot t_j$ and OS overhead O_j are set to zero when the corresponding group contains no critical task. Finally, as discussed in Section 3.2.3, only the communication latency $L_k D_{i_1i_2}$ is zero when the corresponding edge is not on critical path, whereas its contribution $\tau_k p_k B_{i_1, i_2}$ to network arbitration overhead is always captured independent of the synthesis mode.

The weights in (4.21) cannot be simply rounded up as was the case for constraints because doing so disrupts the cost structure of a problem unless small time units are used. Using such units can result into huge numbers which can easily overflow the computation of the value of the objective function.

Ignoring the contribution of small weights before rounding can avoid severe disruptions of the cost structure. A problem with this approach is that leaving large weights only in (4.21) most likely will lead to bad problem formulations. For instance, if a problem instance has a very slow network resulting into a weight with a large magnitude, then ignoring much smaller weights for execution times on

processors is meaningless because we can easily have a case where that slow network is not allocated at all. That would lead to poor architectures because task mappings would be insensitive to execution costs.

A better approach would be to eliminate large weights instead under the assumption that those will not appear in the optimum solution anyway, effectively allowing us to prune the design space before resolution. However, that requires us to prove before hand that eliminating a weight will not constrain the design space. It is currently not clear how that could be done generally in presence of various mapping constraints.

Therefore, the weights are converted into processor cycles so that (4.21) matches (3.47) as close as possible. In order to avoid large numbers, these weights are expressed in terms of cycles that would have been spent on the slowest processor, normalized by the number of cycles on the same processor that would have been consumed for the smallest weight in the objective.

Even with this measure, overflows can still occur if the problem has a sufficiently large number of variables. Unfortunately, one cannot determine at formulation time with absolute certainty if overflows can occur because it is not known a priori if a particular truth-value assignment that is known to overflow will actually be encountered during resolution. This is a major limitation of this method.

4.4 Comparison of Synthesis Results

The same design flow of Figure 3.1 was used to study the effectiveness of this SAT-based synthesis after extending the implemented tool (problem formulator in Figure 3.1) to support Clingo and its language. This section compares synthesis results obtained, against those in Chapter 3, starting with the six non-realtime applications, followed by preemptive scheduling for WLAN and WCDMA, and finally makespan optimizations for WLAN.

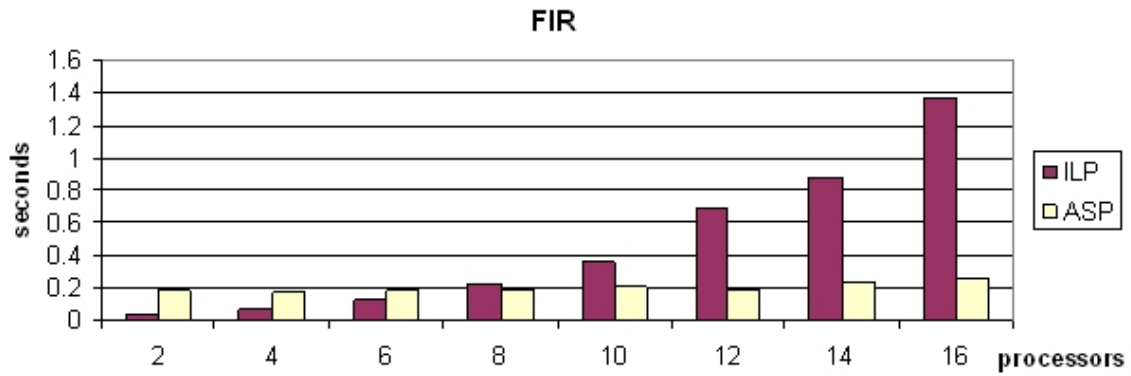
Whereas good solver settings for `lp_solve` were obtained and used through try and error in chapter 3 (node auto-ordering, most feasible basis crash, automatic branch & bound branching, and pre-solving of rows and columns), no options were specified for ASP/Clingo. Still, we determined that splitting the objective into its three constituent parts for execution, communication and scheduling cost significantly speeds up clasp runtime by up to two orders of magnitude. This circumstance was exploited subsequently since the speed up was not accompanied by any penalty in quality of the solution. Splitting the objective function is a feature in the language supported by Clingo that was conceived to avoid possible overflows when computing the value of the objective function because of integrality constraint for weights [105].

4.4.1 Non-Realtime Applications

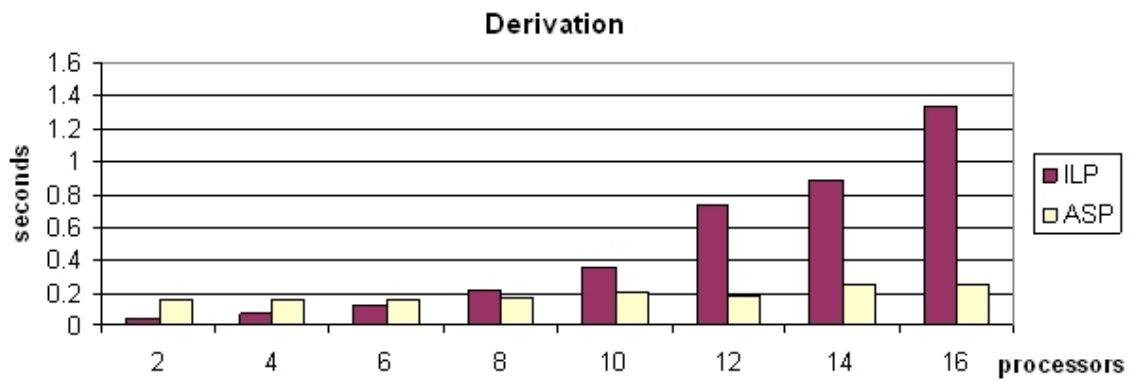
As in Chapter 3, we started with 4 tasks and 2 networks, and then progressively increased the number of processors from 2 to 16. Then the number of processors was fixed at 16, and that of networks was progressively increased to 5. Finally, the number of tasks was increased to 22. The parameters of used processors and networks were the same as given in Tables 3.4 and 3.5 respectively. Also in this case, the synthesis machine had a 1.66GHz T5500 processor and 2048M of memory.

Figure 4.3(a) gives the comparison of ASP-based synthesis versus ILP for the six applications when the number of processors is increased. For small number of processors, ILP outperforms ASP-based synthesis. However, considering that those are rather simple problems, the difference is really insignificant. As problems become more complex, the runtime for ASP-based synthesis does not grow exponentially, thereby outperforming ILP. This evidence that the ASP-based synthesis is better is further cemented in other parts of this section.

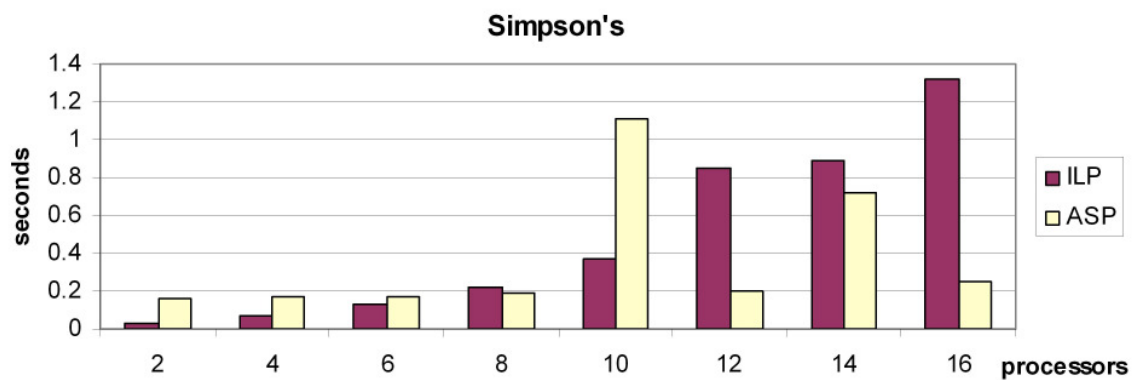
Figure 4.4(a) shows what happens when the number of network is increased. While both methods



(a) FIR

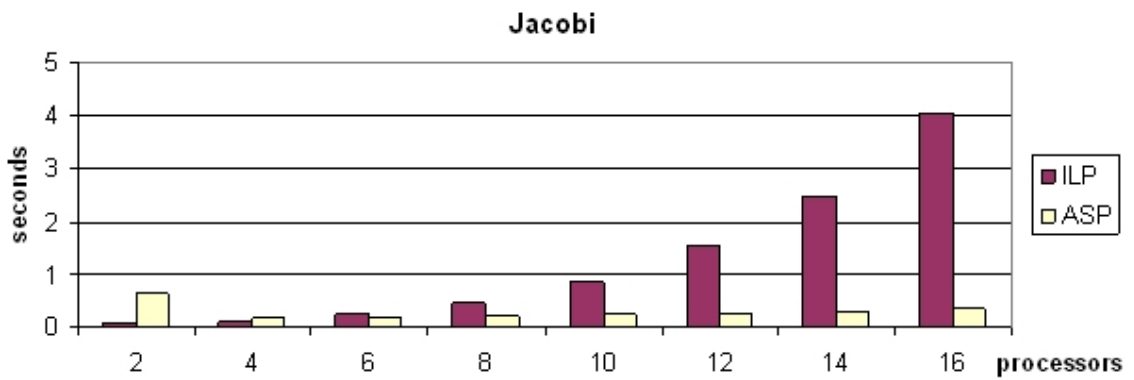


(b) Derivation

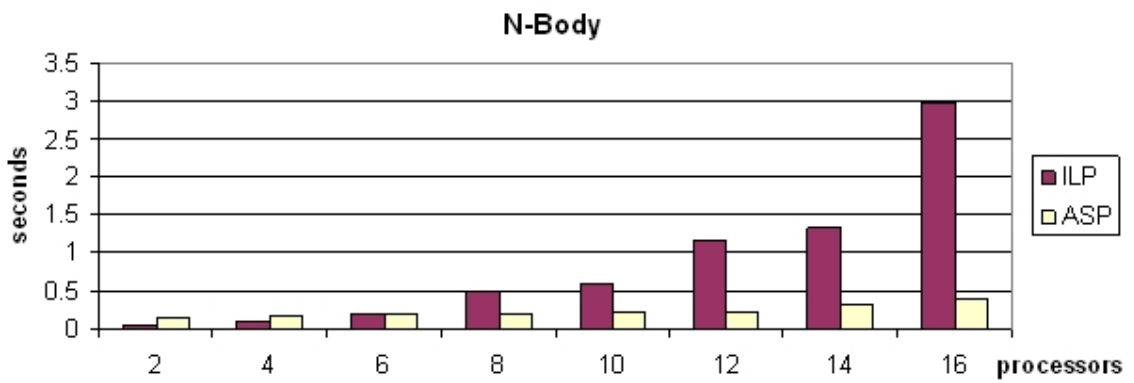


(c) Simpsons

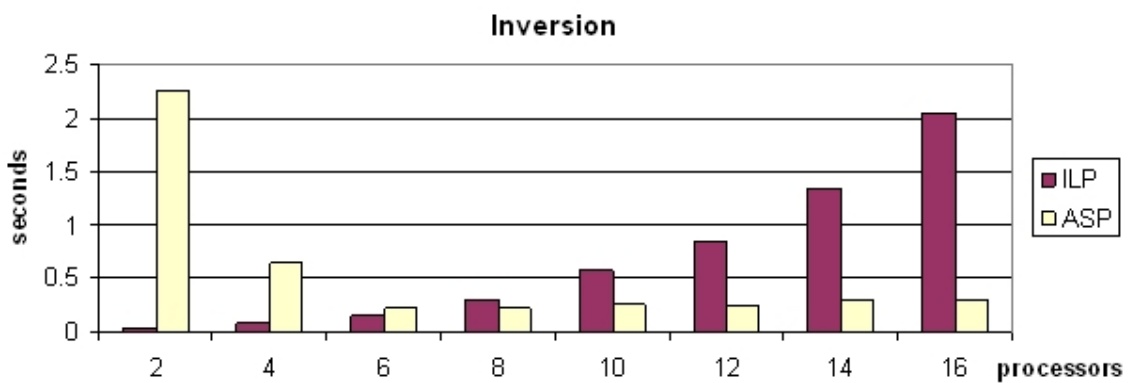
Figure 4.3: Continued on next page



(a) Jacobi



(b) N-Body



(c) Inversion

Figure 4.3: ASP-ILP Comparison : Increasing the number of processors

have nearly a constant runtime, ASP-based synthesis was up to 8 times faster with only one exception where the runtime between the two was comparable.

Increasing the number of tasks on the other hand disproportionately increases the problem size, thereby posing a challenge for both approaches. Figure 4.5(a) depicts the result. As in Chapter 3, a timeout value of 28800 seconds was used. All columns which terminate at that boundary in the figure indicate that a suboptimum solution was found. Columns which exceed the boundary, i.e. those which touch the 100,000 seconds line, indicate that no solution was found by timeout. Otherwise, optimum solutions were obtained. As before, the Jacobi application is missing because its MPI implementation is not scalable with the number of tasks.

The figure shows that ASP-based synthesis outperforms ILP for $4 \leq N \leq 12$. Beyond that region, ILP-based synthesis is either faster, or at least finds a suboptimum result by timeout. A closer look however showed that when ASP-based synthesis performs badly, then most of the time is spent reading ASP programs. In fact, in all cases where Clingo timed out, clasp did not get any chance to execute at all.

The reason was that, when reading large ASP programs generated for higher number of tasks, the system was running out of memory resulting to heavy thrashing to the extent that the machine became unresponsive. It was thus not possible to conclude in these cases whether core SAT-based methods (i.e. conflict-driven learning, unit propagation, etc) were actually effective.

Because scheduling analysis carries the largest share of responsibility for problem size explosion, it is likely that integrating this step in the solver would reduce memory requirements, probably exposing that SAT-methods are indeed very effective given the evidence elsewhere in this section. Without such an integration, $N > 14$ appears to mark the limit where problems become impractical to solve. This is the reason as to why it is vital to consider application-specific heuristic which do not require a full schedulability analysis, a topic which is discussed in detail in Chapter 5.

4.4.2 Realtime Preemptive Scheduling

This subsection gives a comparison of synthesis results obtained for ASP and ILP-based synthesis flow. As in section 3.2.2, the applications used are the signal processing chains for WLAN and WCDMA for the same processors and networks as detailed in Tables 3.4 and 3.5.

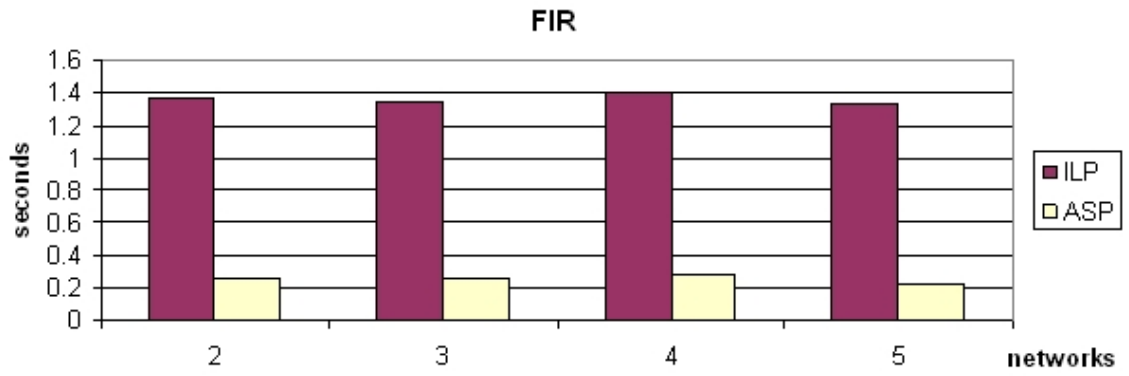
As in Section 3.2.2.6, we additionally compare the synthesis run time with and without fixed-priority preemptive scheduling using configurations with a basic cyclic executive and with a preemptive kernel. Table 4.1 summarizes the comparison.

The columns “# Cons.” show the number of constraints and the number of inference rules for ILP and ASP modes respectively. Similarly, the columns “# Var.” show the number of decision variables and atoms for the two modes. These numbers give a measure for the complexity of the problem instances. The number of variables for ILP mode was much less than the number of atoms for ASP mode because `lp_solve` has a pre-solve option that can reduce the size of the problem by eliminating redundant constraints¹. This option was exploited because pre-solving tends to reduce the solver time.

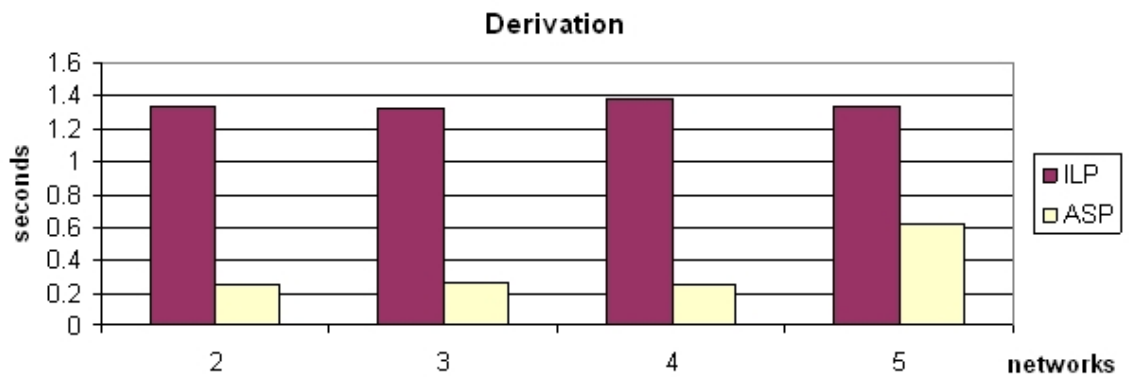
The columns “Form.” and “Solver” show synthesis runtime spent formulating and solving the problem respectively. Formulator time is rather large where a large percent of the time is spent reading text files generated from MPI simulations². The sizes of these files in these experiments were 4.8GB and 6.4GB for WCDMA and WLAN respectively. They contain, among others, time stamps for each data packet transmitted between tasks. This large time is not a limitation for automated exploration: much faster time can be achieved by using compressed binary files and/or usage of cache files to store

¹This comparison was omitted in the previous subsection due to the large number synthesis scenarios. The general trend was the same.

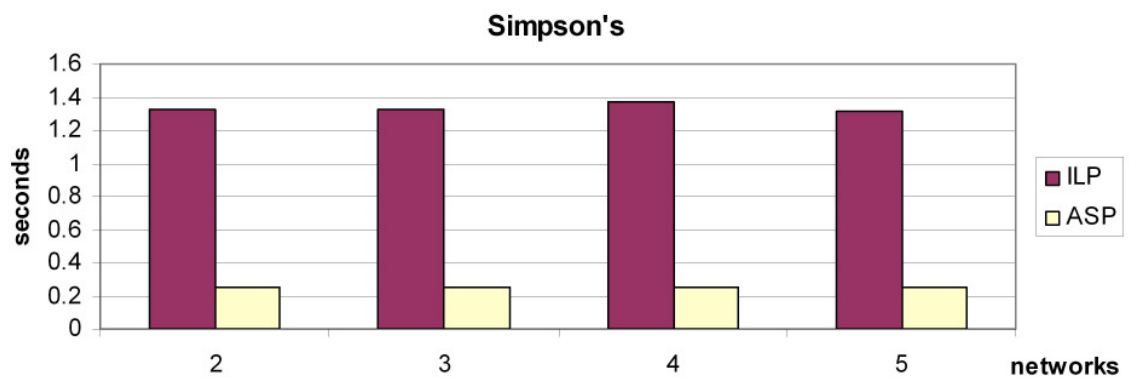
²The formulator time was excluded in the comparison in the previous subsection. These are given here for a more complete picture.



(a) FIR

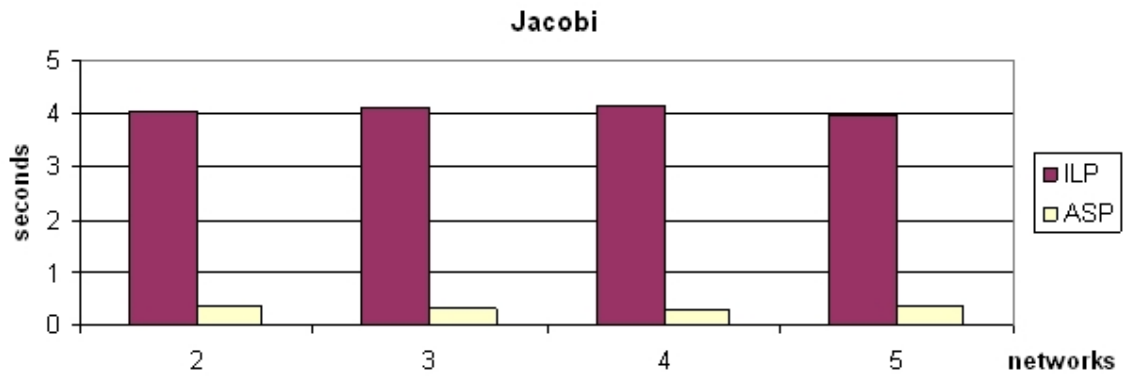


(b) Derivation

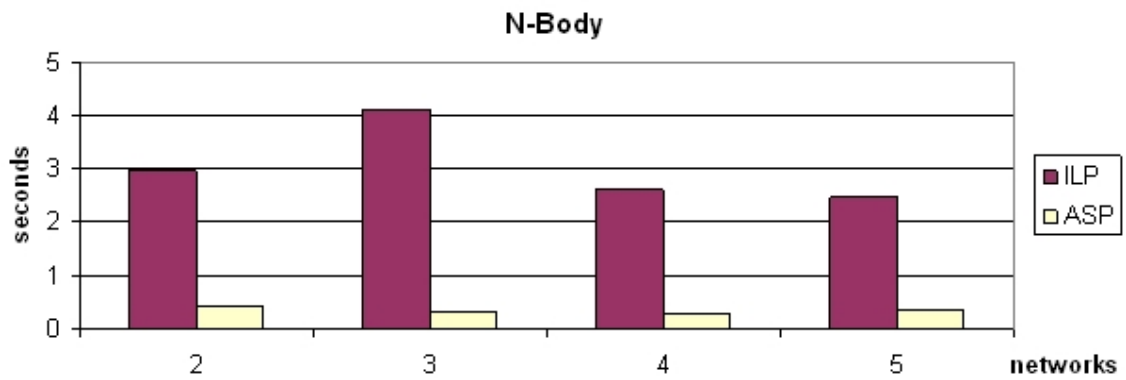


(c) Simpsons

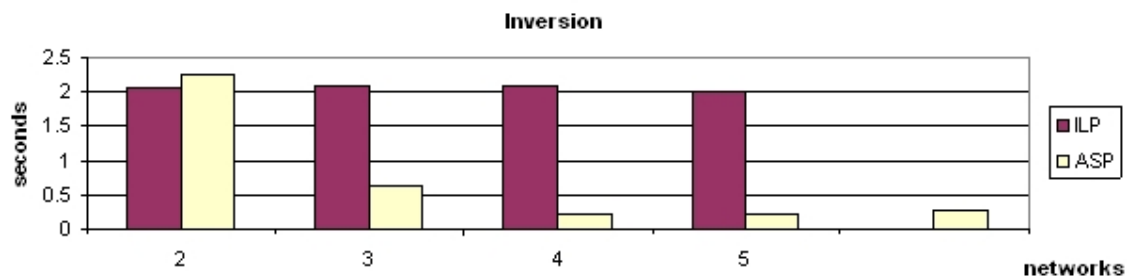
Figure 4.4: Continued on next page



(a) Jacobi

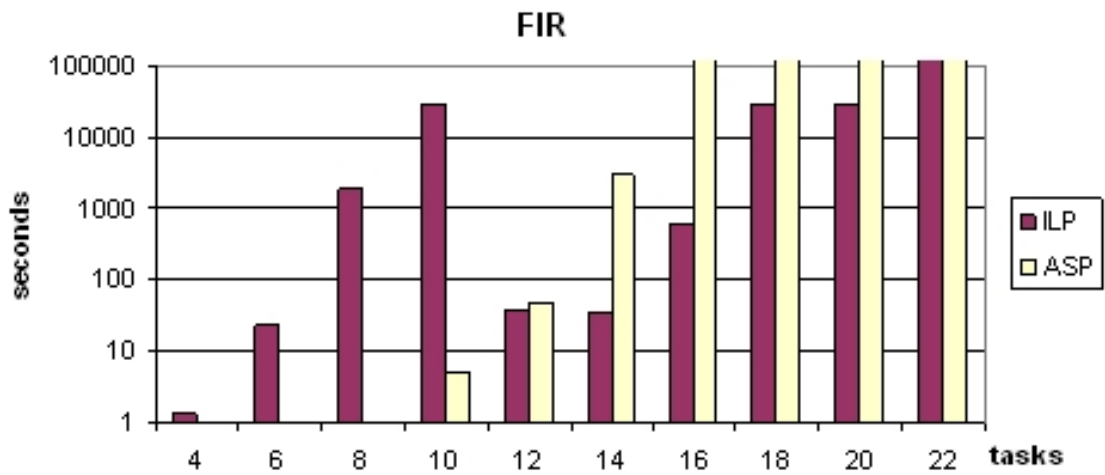


(b) N-Body

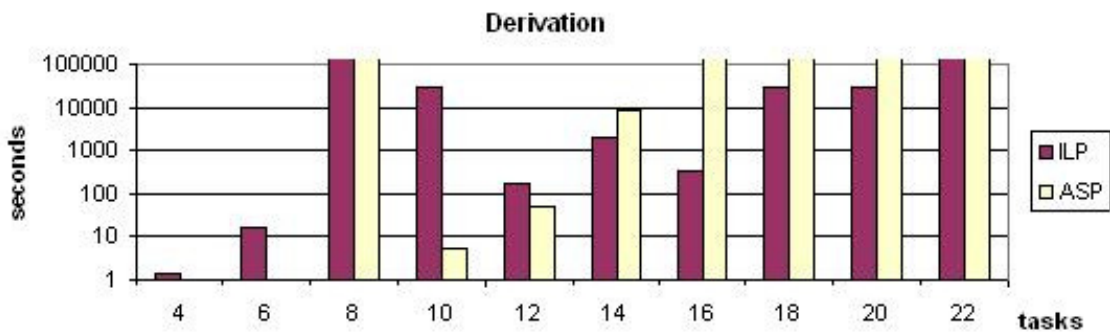


(c) Inversion

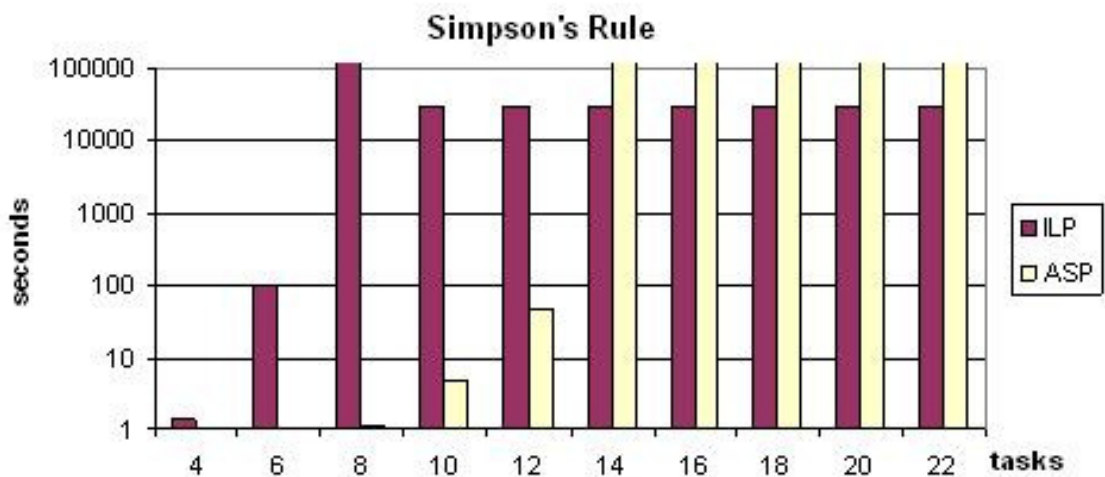
Figure 4.4: ASP-ILP Comparison : Increasing the number of networks



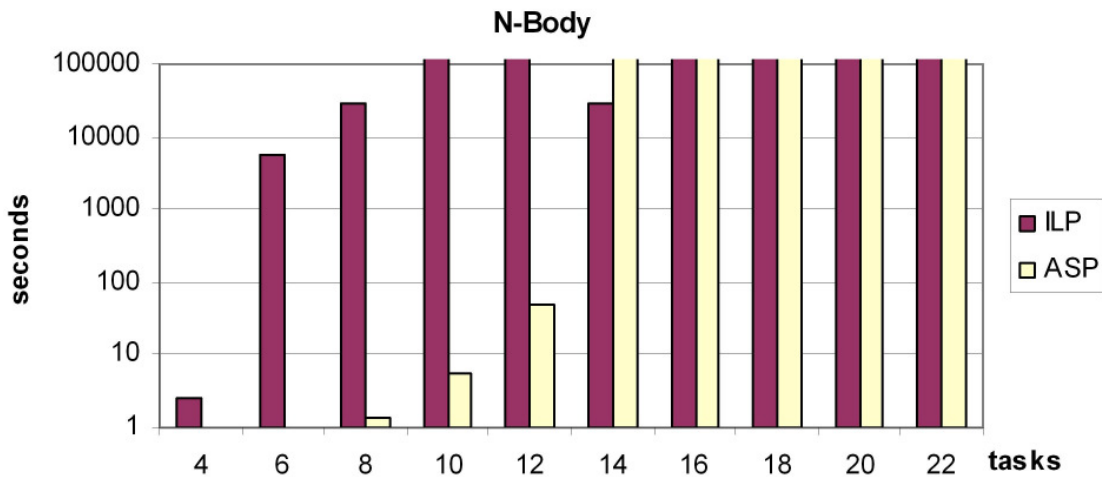
(a) FIR



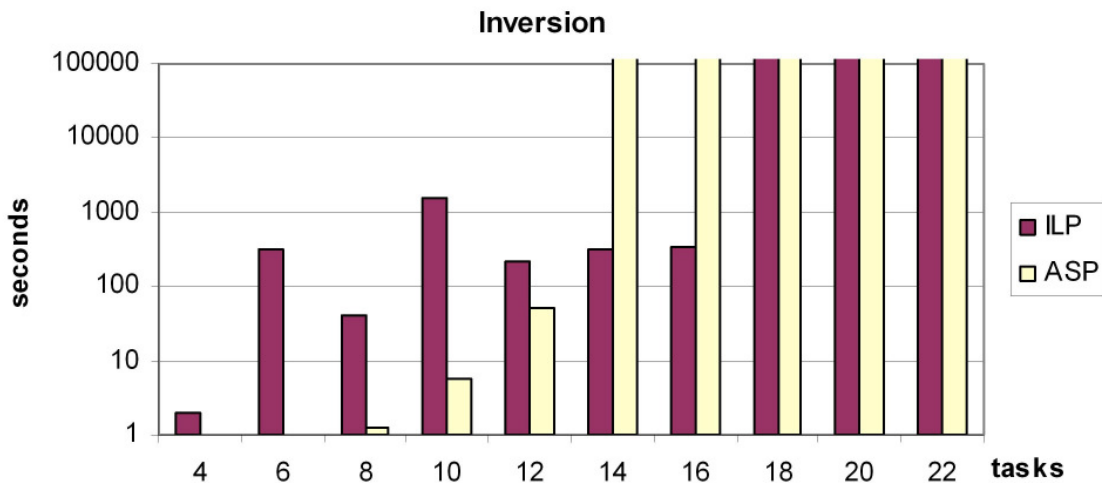
(b) Derivation



(c) Simpsons



(a) N-Body



(b) Inversion

Figure 4.5: ASP-ILP Comparison : Increasing the number of tasks

frequently used information only. The latter technique is typically used in synthesis tools to avoid expensive redundant computations or analysis.

As is evident from the table, the solver times for ASP-based synthesis are dramatically shorter by up to three order of magnitude, again cementing the evidence that core SAT-techniques are effective for this synthesis problem. Note that, as before, the value 28811 for WCDMA under cyclic scheduling indicates a timeout.

The columns “Obj.” show the value of the objective function after optimizations. Comparing the value of the objectives for the two modes, the differences before rounding are insignificant with the exception of the timeout case where ASP mode found a better solution. The impact of more restrictive constraints for ASP-based synthesis due to rounding as previously discussed was not apparent in these experiments. While these particular results are suggestive, experiments with a much larger set of parallel programs are still required to characterize the potential impact, particularly when the issue around the thrashing problem has been resolved.

Figure 4.6 shows synthesized architectures under the two modes for preemptive scheduling. These architectures are very similar for WLAN, and the same architecture was obtained for WCDMA. This result emphasizes the potential for ASP-based synthesis, since the quality of results was not traded against solver runtime. The difference between these figures and those in Figure 3.10 is that the former are for the case of reduced number of processors in order to take advantage of a pruned design space as discussed in Section 3.2.2.6.

4.4.3 Makespan Optimization

Finally, Table 4.2 summarizes the comparison of the two synthesis approaches for makespan optimizations. The comparison is for the case of reduced number of processors to take advantage of design space pruning as previously discussed.

Also in this case, ASP-based synthesis was much faster by up to three order of magnitude. As discussed in Section 3.2.3, the limitation for heuristics for the longest path problem is a pitfall: since the computation of the longest path could go wrong, makespan optimizations can proceed under false assumptions concerning what is the critical path.

Only a slight deviation in the computation of the critical paths was obtained for the two approaches under cyclic scheduling as shown in the table. Under preemptive scheduling, the final critical path was the same in both cases. Therefore, the comparison in the table is conclusive in favor of ASP-based synthesis.

Table 4.1: ASP-ILP Comparison under Scheduling (Non-shaded rows for ILP, Shaded rows for ASP)

| | Cyclic | | | | | Preemptive | | | | |
|-------|--------------|--------|----------------|--------|--------|--------------|--------|----------------|--------|-------|
| | Problem size | | Run time (sec) | | Obj. | Problem size | | Run time (sec) | | Obj. |
| Appl. | # Cons. | # Var. | Form. | Solver | (sec) | # Cons. | # Var. | Form. | Solver | (sec) |
| WLAN | 7089 | 3638 | 497 | 636 | 0.0001 | 5901 | 2538 | 552 | 1010 | 0.10 |
| WCDMA | 32261 | 13320 | 701 | 28811 | 18.25 | 25394 | 12792 | 704 | 19567 | 16.90 |
| WLAN | 7033 | 207412 | 1467 | 2.421 | 0.0002 | 7033 | 207412 | 1477 | 16.797 | 0.08 |
| WCDMA | 25631 | 21806 | 977 | 6.781 | 16.898 | 25631 | 21806 | 1125 | 10.219 | 16.90 |

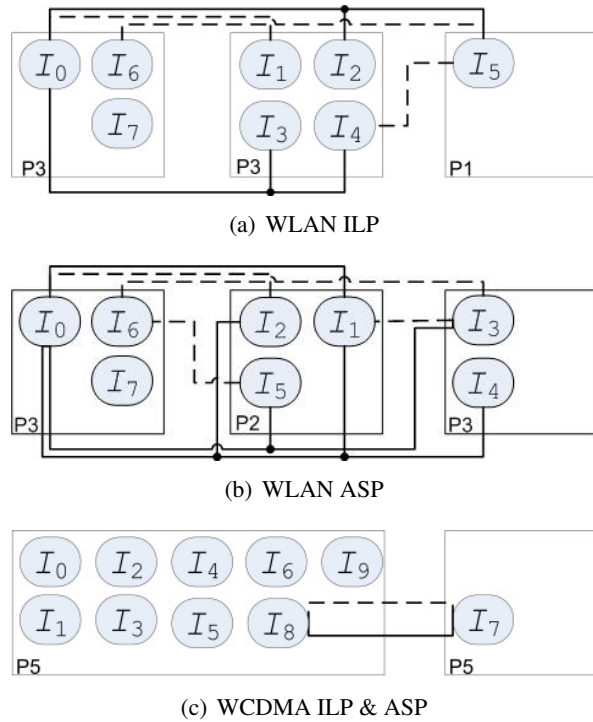


Figure 4.6: Synthesized architectures under preemptive scheduling (dashed lines links, full lines buses)

Table 4.2: ASP-ILP comparison for makespan optimizations (non-shaded row for ILP, shaded row for ASP)

| Cyclic | | | Preemptive | | | | |
|-------------------------|----------------|---------|------------|----------------------|----------------|---------|---------|
| Final CP | Run time (sec) | | Obj. | Problem size | Run time (sec) | | Obj. |
| | form. | solver | (sec) | | form. | solver | (sec) |
| I0 I7 I6 I5 I4 I2 I1 I3 | 2360 | 808.688 | 32.9932 | I0 I7 I6 I5 I4 I1 I3 | 7093 | 5020.72 | 32.9932 |
| I0 I7 I6 I5 I4 I1 I3 | 2385 | 4.656 | 33.0501 | I0 I7 I6 I5 I4 I1 I3 | 2493 | 7.296 | 32.9932 |

4.5 Chapter Summary

This chapter presented a synthesis approach based on Answer Set Programming with the intent of studying how well techniques from the SAT domain can be used to solve large problem instances. Evaluation results have shown a great potential for this approach, with a demonstrated speed up of up to three orders of magnitude.

There are two open areas that need to be addressed before this method can be adopted for large problem instances:

Memory requirements is a challenge. The need for a full schedulability analysis leads to huge textual ASP programs with the consequence that it becomes impractical to solve very large instances because of system memory limitations. This is not a problem with the solver itself because in all instances where synthesis timed out, the solver did not get a chance to execute at all. It is likely that a method that integrates schedulability analysis in the engine will lead to significant improvements.

Integrality constraints is a problem. The need to meet this constraint means that weights in the objective function can have rather large magnitudes. That, coupled with a sufficiently large number of variables, can overflow the computation of the value of the objective value during resolutions, possibly leading to suboptimum architectures.

These open areas should be addressed in future work. The following chapter presents a synthesis method based on application-specific heuristics which do not rely on a full schedulability analysis. The intent is still the same: to find fast way of conducting explorations in presence of very large problem instances.

5 Greedy-Like Heuristics

Experimental results from Chapter 4 provide the evidence that ASP-based synthesis is superior to ILP-based synthesis in terms of runtime without penalties in quality of results. Nevertheless, we face two limitations:

- When the number of tasks N exceeds 14, ASP-based synthesis too takes too long to finish. The fundamental challenge in both cases is the explosion of the number of variables and constraints (atom and rules for ASP-mode) which is accentuated by the need to pre-compute feasible schedules for all possible combinations of task mappings. One possible solution is the integration of problem formulation and resolution as mentioned in Section 3.2.2.4, an approach that requires the development of suitable synthesis algorithms.
- There is an uncertainty as to whether overflows can occur when problem instances contain a certain unknown number of decision variables or atoms due to integrality constraints. While overflows can be detected after synthesis by post computing the value of the objective based on the stable model, it is hard to tell if the solver missed better architectures because of false impressions that certain assignments are low cost.

Therefore, given the need for fast resolutions for large problem instances in order to facilitate explorations, this chapter presents three heuristics and contrasts their runtime and quality of results against ILP-based synthesis. These heuristics allow fast synthesis even for very large problems thereby lifting the first limitation mentioned above. The second limitation remains however for the general synthesis case because heuristics do not necessarily lead to optimum results.

These three heuristics attempt to exploit certain characteristics of the automated synthesis problem which are discussed in subsequent subsections. All three however are rooted to a common idea stemming from experimental results on makespan optimizations in Chapter 3.2.3.

Table 3.16 showed that the value of the objective function after hypothetical allocation of resources is very high even though each node and each edge in the application graph is mapped to the best possible resource for it from the IP library. Further reductions in this value are only possible by minimizing inter-task communication cost through processor sharing. This is not a surprising result because we know that wire delays do not scale down as fast as gate delays so that, in general, inter-task communication costs are dominant. The implication is that, starting with an hypothetical assignment, we cannot further improve task execution and scheduling cost. Further improvements in the overall solution, if any, can only be achieved by eliminating inter-task communication over network resources, implying that a suitable algorithm can arrive at a globally optimum solution beginning with an hypothetical solution by making correct processor sharing decisions. The ideas presented in this chapter attempt to exploit this characteristic of the synthesis problem.

The proposal that a globally optimum solution can be found by making correct sharing decisions suggests that if we can find a way to make a correct decision at each step, then we probably can efficiently solve the synthesis problem using a linear-time greedy algorithm. It is therefore both interesting and important to examine in detail if the synthesis problem does exhibit greedy properties. This is the subject of Section 5.1. Conclusions from the section are used to derive three different heuristics in

Sections 5.2, 5.3 and 5.4. Section 5.5 finally compares synthesis results obtained using the three algorithms in terms of runtime and quality of results, where ILP-based results from chapter 3 are taken as a baseline.

Throughout this chapter, the term “allocation” means that a processor or a network resource is selected to be used for task execution or data transfer, similar to its classical usage in digital circuits synthesis [107]. The term “assignment” signifies that a mapping decision is made, for instance, to map a task on a certain processor. The two processes are equivalent, the distinction is solely made to emphasize that a decision is being made from the perspective of the IP library and from the perspective of the application graph respectively. The notion of “edge elimination” refers to deassigning an edge, and assigning the two corresponding tasks to the same processor. The term “deallocation” is exclusively used to mean that a particular resource is completely removed from the solution set.

5.1 Substructure of the Synthesis Problem

Greedy algorithms solve optimization problems by making best decisions at any given resolution step based on information available at that step, effectively selecting locally optimum choices. Such algorithms are fast because they do not need to backtrack. They may find globally optimum solutions, very good suboptimal solutions, or very bad solutions. Problems which can be well solved are those which have two properties: optimal substructure, and greedy choice property[2].

A problem has optimal substructure if the problem consists of subproblems such that an optimum solution to the problem contains optimal solutions to the subproblems. A solution to such problems consists of making choices. When a choice is made, one or more subproblems remain for which choices needs to be made. The greedy choice property is exhibited when at each decision step the best choice can be selected without considering results from subproblems such that a globally optimum solution results[2].

Our synthesis problem at hand is solved by making mapping and allocation decisions. Each choice affect either the total execution time $\sum T_{ij}$, the switching overhead T_{switch} or the communication overhead T_{net} according to (3.47). If a decision is made to allocate a certain processor, then resulting subproblems are

1. Which group of tasks should be mapped to that processor,
2. How the tasks should be scheduled, and
3. Which communication resources should be used for exchanging data between tasks on that processor and others on different processors.

However, in order to fulfill the first greedy requirement, if at any step a processor J_j is allocated, then

- a) J_j must be a constituent component of the optimum architecture,
- b) A group G_l mapped to J_j must have the lowest contribution to the sum $\sum t_{ij}$ and to T_{switch} compared to any other group that can be mapped to that processor, and
- c) The group G_l must result to the lowest contribution to T_{net} in the overall solution due to edges incident on its tasks.

If we were to modify the formulation of the synthesis problem in Chapter 3 to allow the presence of an empty group, and to allow multiple mappings for that empty group, then we could guarantee these three conditions as long as we compute both the sum $\sum T_{ij}$, T_{switch} and T_{net} before committing a mapping decision: i.e. allocate all processors, map empty groups to them, sort tasks according to

normalized execution times, decide where to map each task, allocate the best available communication resource for it if required, and finally remove processors with empty groups only.

Unfortunately, subproblems in this synthesis problem are *generally* not independent because of constraints. Optimum substructure requires subproblems to be independent in the sense that decisions made for any subproblem cannot affect decisions made in any other subproblem [2]. When a mapping decision is committed at any given step, resources are occupied. Generally, the occupation affects decisions for other subproblems, because constraints which needs to be observed now dictates that certain choices cannot be made because of previous choices. Consequently, while a criteria for making best decisions exists, the non-optimum substructure limits its effectiveness. The implication is that we cannot design a greedy algorithm for the *general* problem, and by extension, not even a dynamic programming algorithm.

Therefore, going back to the original problem at the beginning of this chapter, the interesting question is how we can make edge elimination decisions such that we can at least reach a good solution starting from a hypothetical assignment.

It would be interesting at this juncture to examine how well general-purpose optimization methods such as evolutionary (e.g. genetic) and local search algorithms (e.g. simulated annealing, tabu search) can be employed for synthesis. However, selecting any of such algorithms is only meaningful if the structure of the problem is evaluated in the context of the algorithms, and if experiments with a large number of problem instances are conducted [108]. This is an important activity which is here deferred for future work.

Special-purpose algorithms offer another alternative for searching the design space. Such algorithms are specifically designed to exploit the structure of a problem at hand. In the context of the synthesis problem from hypothetical assignment, the question is how edges should be eliminated given the knowledge that making such decisions effectively constrain the design space for dependent subproblems. The rest of this chapter addresses this question.

5.1.1 Instances Without Resource Constraints

When a problem instance has resources with adequate capacity and there is a feasible schedule for all groups on any processor, then we have a special case where the problem exhibits optimum substructure. We begin by defining the notions of resources and adequate capacities as used in the context of this chapter.

Definition 1. *A resource in the synthesis problem is either a physical FPGA, or a processor $J_j \in \mathcal{J}$ or a “communication resource” $C_k \in \mathcal{C}$.*

Definition 2. *A resource has “adequate capacity” if it is used in an optimum solution, and if the usage either does not exhaust its capacity, or if the resource is fully utilized, then the solution would be the same if the resource had a larger capacity.*

In such scenarios, best decisions can be made at each step, meaning that a greedy algorithm exists. Since many problem instances fall under this category, this section provides the proof for greedy characteristics of this special case as a prerequisite for further discussions in this chapter.

For some optimization problems, matroids can be used to determine when greedy algorithms yield optimum solutions [2]. Matroids are combinatorial structures which are defined as follows: a matroid $\mathfrak{M} = (S, L)$ is an ordered pair such that

1. S is a finite nonempty set.
2. L is a nonempty set of subsets of S such that if $B \in L$ and $A \subseteq B$, then $A \in L$. The set L is known as independent subsets of S , whereas the condition is known as hereditary.

3. \mathfrak{M} satisfies the exchange property, meaning that if $A \in L$, $B \in L$, and $|A| < |B|$, then there is an element $\chi \in B - A$ such that $A \cup \{\chi\} \in L$. This means that when χ which is in B but not in A is added to A , then the result is an independent set. In that case, we say that A has been extended by χ .

The independent subset is defined such that a specific property over S is valid. A subset $A \in L$ is called maximal if it has no extensions. The implication is that if an $\chi \in S$ is added to A , then the subset becomes dependent, meaning that the subset loses that specifically defined property.

If there is a function w that assigns a positive weight $w(\chi)$ for each $\chi \in S$, then M is known as a weighted matroid (in analog to a weighted graph). Some optimization problems can be formulated as a problem of finding a maximum-weight independent subset in a weighted matroid. For such problems, greedy algorithms lead to optimum solutions [2]. It can be shown that matroids exhibits greedy-choice and optimal substructure properties [2]. Conversely, if we can show that a problem can be defined as a matroid, then we have a proof for these two properties for that problem. This is what we do next.

Using the notation from Chapter 3, we note that a solution to the synthesis problem is fully characterized by decision variables $x_{i,j}$ and $z_{ki_1i_2}$. Let S_G be a set containing these decision variables, and L_G be the independent subset such that $A \in L_G$ is a valid full or partial solution. A valid $A \in L_G$ represents a full solution to a problem instance if the mapping for every task and for every edge in the application graph G is covered by a decision variable in the subset. Otherwise, A represent a partial solution. The term “valid” signifies the following:

- i A decision variable $x_{i,j}$ or $z_{ki_1i_2}$ is an element of A if and only if its value is 1 in the solution represented by A .
- ii A solution presented by A does not violate the mapping constraint (3.1) in that a task is mapped only once in A . For instance, the decision variables $x_{1,2}$ and $x_{1,7}$ cannot both be elements of A . However, L_G is independent if $A \in L_G$, $B \in L_G$, $x_{1,2} \in A$ and $x_{1,7} \in B$.
- iii If $A \in L_G$ and $z_{ki_1i_2} \in A$, then there exists a full solution $C \in L_G$ such that $A \subseteq C$, $x_{i_1,j_1} \in C$ and $x_{i_2,j_2} \in C$, meaning that C contains a mapping for the edge between i_1 and i_2 such that (3.13) is satisfied. However, (3.13) is relaxed to include intra-processor communication resources so that the condition $J_{j_1} \neq J_{j_2}$ is dropped. The relaxation does not distort the synthesis problem because the cost of intra-processor communication resources can be appropriately defined.
- iv Similarly to (ii) above, only one $z_{ki_1i_2}$ can exists in $A \in L$ for any given edge. However, L_G is independent if for example $A \in L_G$, $B \in L_G$, $z_{112} \in A$ and $z_{212} \in B$.

Theorem 1. *If resources in a synthesis problem have adequate capacity and there is a feasible schedule for a group $G_l = \mathcal{I}$ on any processor in \mathcal{J} , then $\mathfrak{M}_G = (S_G, L_G)$ is a matroid.*

Proof. With the foregoing definition, it follows that S_G is non-empty provided there is a solution to the synthesis problem. Also, L_G is hereditary because if $B \in L_G$ is a valid full or partial solution, then $A \subseteq B$ is also a valid full or partial solution because A necessarily contains decision variables from B such that (i)-(iv) above are satisfied.

Furthermore, suppose $A \in L_G$, $B \in L_G$ and $|A| < |B|$. To satisfy the exchange property, there must exists a decision variable $\chi \in B - A$ that can extend A such that L_G remains independent. The case where $A \subset B$ is trivial. If on the other hand that is not the case, we distinguish between the following cases:

- Both A and B contain decision variables for tasks only. Since B is larger, then B contains a decision variable for a task not covered in A because of definition (ii) above. It also follows

from this definition that A can be extended by the variable such that the result is an independent subset.

- Both A and B contain decision variables for edges only. Similarly to the case above, A can be extended by some $z_{ki_1i_2}$ from B .
- For all other cases, the minimum difference between A and B is either
 - A contains a single decision variable $x_{i_1j_1}$, and B contains two decision variables $x_{i_1j_2}$ and $z_{ki_1i_2}$. Here, A can be extended by $z_{ki_1i_2}$. This is illustrated as follows:
 - * The two task-mapping decision variables must be different, otherwise $A \subset B$ holds.
 - * The two task-mapping decision variables must be for the same task, otherwise we have a simple extension of the first case.
 - * The minimum difference must thus be for two candidate processors for the same task, therefore A can be extended by a decision variable for an edge (or equally well for another task).
 - A contains a single decision variable $z_{k_1i_1i_2}$, and B contains two decision variables $z_{k_2i_1i_2}$ and either x_{i_1j} or x_{i_2j} . Here, A can be extended by x_{i_1j} (or x_{i_2j}) similarly to the case above.

In either case, mapping constraints according to definitions (ii) and (iv) are not violated, and L_G remains independent.

- A and B exhibit more differences than the minimal case above. The extension above trivially applies.

Therefore, \mathfrak{M}_G satisfies the exchange property, completing the proof that \mathfrak{M}_G is a matroid. \square

Corollary 1. *If A is a full solution, then A is a maximal independent subset in \mathfrak{M}_G .*

Proof. This readily follows from the theorem above. If A is a full solution, there is no extension for A that will not violate mapping constraints. Also, note that because of the relaxation for (3.13) above, all maximal independent subsets of \mathfrak{M}_G have the same size, which agrees with the theorem that all maximal independent subsets in a matroid \mathfrak{M} have the same size [2]. \square

A generic greedy algorithm for finding a maximum-weight independent subset in a matroid proceeds by sorting weights of the elements in S in a monotonically decreasing order [2] as shown in Algorithm 5. Then, for each element in the sorted order, the algorithm extends an initially empty subset A by the element if the result is an independent subset.

```

1: GREEDY( $\mathfrak{M}, w$ )
2:  $A = \{\}$ 
3: sort  $S[\mathfrak{M}]$  into monotonically decreasing order by weight  $w$ 
4: for each  $\chi \in S[\mathfrak{M}]$ , taken in monotonically decreasing order by weight  $w(\chi)$  do
5:   if  $A \cup \{\chi\} \in L[\mathfrak{M}]$  then
6:      $A = A \cup \{\chi\}$ 
7:   end if
8: end for
9: return  $A$ 

```

Algorithm 5: Generic greedy Algorithm [2]

Thus, a greedy algorithm maximizes the quantity $w(A)$

$$w(A) = \sum_{\chi \in A} w(\chi) \quad (5.1)$$

In a synthesis problem corresponding to \mathfrak{M}_G , the weights $w(\chi)$ are the quantities $(T_{ij} + T_{switch_i})$ and $(L_k D_{i_1 i_2} + \tau_k p_k B_{i_1 i_2})$ for $\chi = x_{ij}$ and $\chi = z_{ki_1 i_2}$ respectively according to (3.47), (3.45) and (3.48). Since we want to minimize rather than maximize $w(A)$, the inverse of the weights should be taken to find a lowest cost solution. Therefore, a greedy algorithm can find an optimum solution for a synthesis problem with no resource constraints.

5.1.2 Practical Challenges and Solutions

Note that T_{switch_i} is not a static value when synthesizing in realtime mode because it depends on the response r_i of a task which in turn depends on which other tasks are mapped on the same processor. This raises the question on how we can sort $S_G[M_G]$ when the values of T_{switch_i} are not known a priori.

One possible solution is to redefine S_G and L_G to include the parameters γ_{lj} , and to define weights $w(\chi = \gamma_{lj})$ based on (3.51). The downside of this approach is that schedules for all possible group mappings needs to be pre-computed. As noted in previous chapters, we need to avoid a full schedulability analysis.

In order to avoid the expensive full analysis, the heuristics described herein do the following:

- It is assumed that if $T_{i_1 j} > T_{i_2 j}$, then the task I_1 has a larger response compared to I_2 , with responses in the sense of (3.44). This is a reasonable assumption because tasks with shorter durations tend to have higher periods, and thus higher priorities and shorter responses. Under these circumstances, it is sufficient to sort weights associated with I_1 and I_2 using parameters T_{ij} only ignoring T_{switch_1} and T_{switch_2} according to (3.45), because the relative order of the weights remains the same when scheduling overhead is ignored (i.e. $x_{i_1 j}$ would still appear before $x_{i_2 j}$).
- It is assumed that the overhead T_{switch_i} for any task on any processor is small compared to the execution time of the task on any processor. This is also a reasonable assumption because it would otherwise be impractical to preempt the task. Under these circumstances, the order between $x_{i_1 j_1}$ and $x_{i_2 j_2}$ is arbitrary when the overhead is ignored. Consequently, if the overhead on J_1 is larger, but $x_{i_2 j_2}$ appears before $x_{i_1 j_1}$ in the sorted list, and Line 5 in Algorithm 5 evaluates to true for $x_{i_2 j_2}$, then the algorithm makes the wrong decision. Nevertheless, because the assumption is that the overhead is small, the overall impact to the solution should be insignificant.
- It is speculated that when I_1 and I_2 with $T_{i_1 j_1} > T_{i_2 j_2}$ are mapped on different processors, then the response $r_{i_1 j_1}$ of I_1 on a processor J_1 is large enough compared to the response $r_{i_2 j_2}$ of I_2 on another processor J_2 such that $T_{switch_1} > T_{switch_2}$. This again allows us to sort according to parameters T_{ij} only. However, there is a real possibility that inequality above does not hold for a certain pair of tasks with the consequence that a bad decision can be made if $x_{i_2 j_2}$ wrongly appears before $x_{i_1 j_1}$ in the list. Similarly to the case above, the impact should be minimum assuming relatively small scheduling overhead.

The result is that $S_G[M_G]$ is a partial list which excludes some elements. Algorithm operating on this partial list are greedy-like heuristics, trading optimality for space efficiency. We show later that while such heuristics can lead to poor architectures, a combination of measures from different heuristics does lead to good results.

5.1.3 Instances with Resource Constraints

When a problem instance has resources with inadequate capacity, or when $G_l = \mathcal{I}$ has no feasible schedule for any processor in \mathcal{J} , then the problem lacks optimum substructure as previously discussed. Committing a choice at any decision step may prevent a heuristic from finding an overall better solution due to dependent subproblems. Since it is not known a priori how any edge elimination decision impacts the overall solution, the heuristic must monitor changes in the value of the objective value and backtrack when necessary. Backtracking is the second ingredient in the first heuristic presented in Section 5.2. Because the cost in runtime for backtracking is high, we limit the procedure to one step.

Since there is no strict mapping order, an alternative to backtracking is to restart local searching. In this case, restarting is akin to “jumping” in the design space, potentially undoing bad decisions for a chance of making better ones. Jumping has a random character, and can be beneficial since the heuristic cannot know if making elimination decisions in a different order is wiser. This strategy is employed in the second heuristic which is presented in Section 5.3. Learning which sequences of decisions lead to bad solutions is a better strategy. However, we can expect the runtime for a learning strategy to be comparable to that obtained by ASP-based synthesis since the technique would be similar in nature to conflict-driven learning employed in the solver clasp.

Figure 5.1 depicts the overall structure of the three heuristics, which we discuss below.

5.2 Replace and Search Heuristic

The main idea behind this heuristic is to use a combination of finding a maximum-weight independent subset in a matroid M_G and a backtracking strategy.

The implementation begins with an hypothetical allocation of resources in order to obtain the lowest possible cost for each node and edge as shown in Algorithm 6. The hypothetical assignment proceeds in the same way as in the makespan algorithm in Chapter 3.2.3. This allocation is used to determine which nodes and edges are more critical in initial actual resource assignments in Lines 4 and 6. The actual assignments allocates resources according to their availability and capacity in a greedy fashion. The sorting in lines 3 and 5 corresponds to considering weights in a monotonically decreasing order as described in Section 5.1.1.

Since the going assumption is that synthesis needs to be approached from a communication-centric perspective as mentioned at the beginning of this chapter, processors are allocated first followed by network resources. Initial assignments are followed by an optimization loop that attempts to perform edge eliminations, also in a greedy fashion. There is finally a pass that attempts to fit the design into an FPGA. These steps are explained in detail in subsequent subsections.

The heuristic as a whole is however not strictly greedy in the sense of the generic Algorithm 5 because tasks are assigned first, followed by edges, meaning that weights are not considered in a monotonically decreasing order. The rationale is that the optimization loop leads to good solutions by searching for better solutions starting with initial assignments, where the latter are local optima. Comparison of synthesis results in Section 5.5 demonstrates that this approach is superior to considering all weights in a monotonically decreasing order. Here, the heuristic attempts to exploit Corollary 2 below by edge elimination for the first problem class:

Corollary 2. *If A is a maximal independent subset in M_G , and B is any maximum-weight independent subset in M_G such that $w(A) < w(B)$, then there exist one or several elements $\chi \in S_G$ which when substituted into A such that A remains a maximal independent subset, the quantity $w(A)$ approaches or becomes equal to $w(B)$.*

Proof. The proof follows readily from the definition of M_G . If no such element $\chi \in S_G$ can be

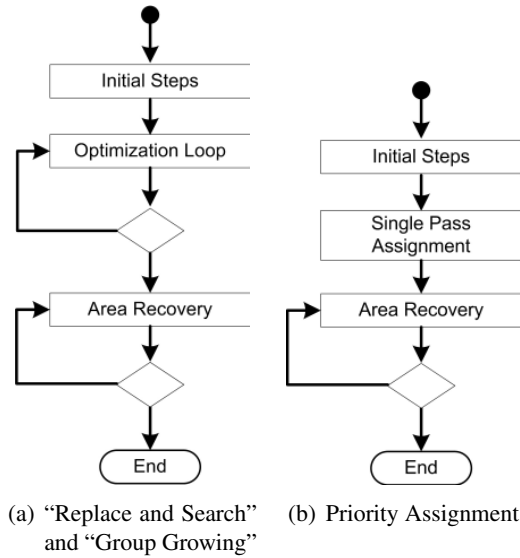


Figure 5.1: Overall structure of synthesis heuristics

found, then A must be a maximum-weight independent subset which contradicts the statement that $w(A) < w(B)$. \square

It therefore follows that we can always arrive at an optimum solution starting with an hypothetical one through incremental substitutions provided the problem instance has no resource constraints. Whereas searching for elements for substitutions is by itself trivial because we know associated weights $w(\chi)$, actual substitutions most likely require backtracking because one cannot simply swap elements since we need a valid solution at the end of each substitution. As a result, we need to perform a local search. If backtracking is limited in depth to achieve good runtime, some bad decisions may still remain in the final solution, thereby leading to inferior architectures despite Corollary 2 above.

Because the heuristic maybe called iteratively in makespan mode, we need to prevent previous assignments from getting overwritten. Therefore, hypothetical assignment is conducted only once, and actual resource assignments skip nodes and edges that have been fixed according to Algorithm 3. Note that if the first makespan pass has no resource constraints, then subsequent iterations also don't so that fixing does not ruin optimum substructure property.

This heuristic sits at the heart of the makespan synthesis loop, i.e. at Line 9 in Algorithm 3, replacing the ILP/ASP solver.

```

1: synthesize(  $G, \mathcal{J}, \mathcal{C}, CP, A_{PE}, A_{net}$  )
2:  $sol = \text{hypotheticalAssignment}(G, \mathcal{J}, \mathcal{C})$ 
3:  $cost = \text{sortNodeCosts}(sol)$ 
4:  $sol = \text{initialTaskAssignment}(G, \mathcal{J}, cost, CP)$ 
5:  $cost = \text{sortEdgeCosts}(sol)$ 
6:  $sol = \text{initialEdgeAssignment}(sol, \mathcal{C}, cost, CP)$ 
7:  $sol = \text{optimizeBacktrack}(sol, CP, \mathcal{J}, \mathcal{C})$ 
8:  $sol = \text{satisfyArea}(sol, A_{PE}, A_{net}, \mathcal{J}, \mathcal{C}, CP)$ 
9: return  $sol$ 

```

Algorithm 6: Heuristic synthesis main steps

5.2.1 Processor Assignment

Algorithm 7 assign tasks one after the other in order of monotonically decreasing weights, where weights are hypothetical task execution times. Since during hypothetical assignment the best processor is allocated for each task, this order considers the most expensive tasks first independent of variations in parameters T_{ij} for execution time on different processors. If synthesis is in makespan mode, fixed tasks are skipped to facilitate multi-pass synthesis. Furthermore, tasks in a critical path CP are considered first even when they have lower execution cost compared to non-critical tasks in order to avoid having latter tasks block best resources which are desired for critical path tasks. Line 5 performs the actual allocation as explained below. A problem is declared infeasible if the allocation fails for one of two reasons as shown in Algorithm 8.

```

1: initialTaskAssignment(  $G, \mathcal{J}, cost, CP$  )
2: for each element in  $cost$  corresponding to task  $I_i \in G$  in  $CP$  do
3:   remove the element from  $cost$ 
4:   if corresponding task is not fixed then
5:     allocateProcessorForGroup( $I_i, \mathcal{J}, CP$ )
6:     if  $I_i$  is not mapped then
7:       report problem not feasible
8:     end if
9:   end if
10: end for
11: repeat lines 3-9 for each remaining element in  $cost$  corresponding to non-critical task
12: return  $sol$ 

```

Algorithm 7: Initial processor assignment

Therein, allocation of a processor for a group of tasks G_l (including for groups with one task only) proceeds by finding a processor in the set \mathcal{J} for which the sum of execution time of all tasks and the scheduling overhead due to G_l and other tasks already mapped on the processor is the smallest.

Line 4 in the algorithm computes the sum of execution time, taking the critical path into account according to (3.47): the coefficient X_i has the value one when the task is in CP , or when synthesizing in non-makespan mode. Line 5 computes the scheduling overhead according to (3.35) using the number of task switching T'_{sl^*j} , context switching overhead t_j , and the OS overhead O_j . The number of switching is computed for a group $G_{l^*} = G_l \cup G_{l_p}$, where G_l is the group for which the allocation is being conducted, and G_{l_p} is a group which is already mapped on J_j . The number of switching is determined for a reduced group if synthesis is in makespan mode in order to capture scheduling overhead for tasks on CP only, otherwise the complete overhead due to the group G_{l^*} is used as discussed in Section 3.2.3.

5.2.2 Network Assignment

Algorithm 9 for initial edge assignment is structurally similar to initial task assignment presented in the previous subsection: edges in the critical path CP are considered first in order of monotonically decreasing hypothetical costs. The first condition in the if-block safeguards against ruining previously assigned edges in case of a multi-pass makespan synthesis mode. The actual assignment in Line 5 is outlined in detail in Algorithm 11 which takes as arguments an edge to be assigned, the set C containing network resources, the list of non-assigned edges excluding the current one, and the critical path. We first explain how a network resource is allocated, because this is used in other algorithms, including Algorithm 11.

Allocation of a network resources is conducted by searching for a best resource as shown in Algorithm

```

1: allocateProcessorForGroup(  $G_l, \mathcal{J}, CP$  )
2:  $time = -\infty$ 
3: for each processor  $J_j \in \mathcal{J}$  do
4:    $t = \sum_{I_i \in G_l} T_{ij} \cdot X_i$ 
5:    $t = t + (T'_{s_{l^*j}} \cdot t_j + O_j)$ 
6:   if ( $t < time$ ) AND (capacity not exceeded) AND (schedule is feasible) then
7:      $time = t$ 
8:     temporary assignment for  $G_l$  is  $J_j$ 
9:   end if
10: end for
11: if  $G_l$  is not temporarily assigned then
12:   report group not feasible
13: else
14:   make assignment permanent
15: end if
16: return  $sol$ 

```

Algorithm 8: Allocating a processor for a group of tasks

```

1: initialEdgeAssignment(  $G, \mathcal{C}, list, CP$  )
2: for each element in  $list$  corresponding to an edge  $I_{i_1} \triangleleft I_{i_2}$  in  $CP$  do
3:   remove the element from  $list$ 
4:   if ( $I_{i_1} \triangleleft I_{i_2}$  is not fixed) AND ( $I_{i_1}$  and  $I_{i_2}$  are not on same processor) then
5:      $netAssign(I_{i_1} \triangleleft I_{i_2}, \mathcal{C}, list, CP)$ 
6:   end if
7: end for
8: repeat 3-6 for all other edges
9: return  $sol$ 

```

Algorithm 9: Initial network assignment

10. Therein, Line 4 calculates the cost for the edge if it would be mapped on a specific network based on (3.48). The cost is the transfer latency for that edge on the network plus the contention overhead due to the edge and all other edges already mapped on the network. The edge is reported to be non-mapped if there is no network resource for which the capacity is already exhausted. Note that, in contrast to processor allocation, a failure to allocate an edge does not immediately mean that a problem is infeasible because the two tasks maybe mapped on the same processor, eliminating a need for a network resource. Moreover, a failure does not necessarily imply that there are no adequate resources for the problem instance since mapping the two tasks to the same processor may lead to a better solution.

Edge assignment in Algorithm 11 begins by an attempt to allocate a resource. If there is no free resource, the allocation fails, the algorithm deassigns the two tasks and attempt to assign them as a group using Algorithm 8. If no processor can be allocated for the group, then the problem is declared infeasible. Otherwise, the algorithm searches for edges incident on I_{i_1} or I_{i_2} which originate or terminate to tasks on the same processor. When found, those edges are either deassigned or removed from the list containing non-mapped edges. This situation may arise because I_{i_1} and I_{i_2} have been moved to another processor.

```

1: allocateNetwork(  $I_{i_1} \triangleleft I_{i_2}, \mathcal{C}$  )
2:  $time = -\infty$ 
3: for each network  $y_k \in \mathcal{C}$  do
4:    $t = L_k D_{i_1 i_2} Z_{i_1 i_2} + \sum_{I_{i_3}, I_{i_4} | I_{i_3} \triangleleft I_{i_4}} \tau_k p_k B_{i_3, i_4} z'_k$  where  $I_{i_4} \triangleleft I_{i_3}$  is on  $y_k$ 
5:   if ( $t < time$ ) AND (capacity of  $y_k$  is not exceeded) then
6:      $time = t$ 
7:     temporary assignment for the edge  $I_{i_1} \triangleleft I_{i_2}$  is  $y_k$ 
8:   end if
9: end for
10: if edge  $I_{i_1} \triangleleft I_{i_2}$  is not assigned then
11:   report edge not mapped
12: else
13:   make assignment permanent
14: end if
15: return  $sol$ 

```

Algorithm 10: Allocating a network resource for an edge

```

1: netAssign(  $I_{i_1} \triangleleft I_{i_2}, \mathcal{C}, list, CP$  )
2: allocateNetwork( $I_{i_1} \triangleleft I_{i_2}, \mathcal{C}$ )
3: if  $I_{i_1} \triangleleft I_{i_2}$  is not mapped then
4:   deassign  $I_{i_1}$  and  $I_{i_2}$ 
5:    $G_l = \{I_{i_1}, I_{i_2}\}$ 
6:   allocateProcessorForGroup( $G_l, \mathcal{J}, CP$ )
7:   if  $G_l$  is not mapped then
8:     report problem not feasible
9:   else
10:    for each task  $I_i$  on the processor where  $G_l$  is mapped do
11:      if there is an edge between  $I_i$  and  $I_{i_1}$  or  $I_{i_2}$  then
12:        if edge is mapped to a network then
13:          deassign the edge
14:        else
15:          remove the edge from  $list$ 
16:        end if
17:      end if
18:    end for
19:  end if
20: end if
21: return  $sol$ 

```

Algorithm 11: Assigning an edge to a network

5.2.3 Optimization Loop

Initial task and edge assignment attempt to find best resources. The result is a local optimum which can be further improved upon through edge elimination. The optimization loop in Algorithm 12 is designed for that purpose. Edge elimination is also greedy-like, where the weights are inter-task communication costs consisting of the sum of data transfer latency and associated overhead for each edge as shown in Line 2. The cost model is derived from (3.48).

The main loop iteratively attempt to eliminate edges in order of monotonically decreasing costs and terminates when the value of the objective function remains unchanged, when there is no edge left that has not been considered, or when an arbitrarily set number of iterations has been set.

The the first termination criteria is for cases in which the problem instance has resource constraints, either because of schedulability constraints, or because of inadequate resource capacities. Therefore, the optimization loop is effectively a local search algorithm which terminates when a local minimum is detected, the latter being either the initial assignment, or a better solution. The search itself is conducted in the neighborhood of a current step in the iteration as discussed below. The starting point is either the value of the longest path in G or the sum of all costs going into the optimization loop depending on whether the synthesis is in makespan mode. The computation of the longest path is the same as the one described in Section 3.2.3. The third termination criteria prevents endless trials which can otherwise occur because of task movements as it will become apparent in the discussion below.

An attempt to eliminate an edge begins by deassigning the edge and corresponding tasks. Because the intent is to map the two tasks on one processor, an attempt is made to allocate a processor for a group consisting of the two tasks based on Algorithm 8. Two things can happen during the attempt:

- The two tasks cannot be mapped, meaning that the problem instance has resource constraints. Therefore, the iteration proceeds with the next edge not yet considered. Effectively, the search backtracks to a previously found solution and attempts to follow a different path.
- A mapping for the two tasks has been found, either because the problem instance has no resource constraints, or because local search has found a potentially feasible solution. In either case, we must consider what happens to other edges as exemplified in Figure 5.2.

In the example, the algorithm attempts to eliminate the expensive edge $I_2 \prec I_3$, and finds a potentially feasible mapping for the two tasks on processor c. In so doing, the edge $I_0 \prec I_2$ now crosses processor boundaries from a to c. Line 20 in the algorithm attempts to assign edges that became free because of such movements according to Algorithm 10. Two things may go wrong in this new attempt:

- New edges cannot be assigned because of inadequate capacity, meaning that eliminating a specific edge is not feasible in the current step (but not necessarily generally infeasible).

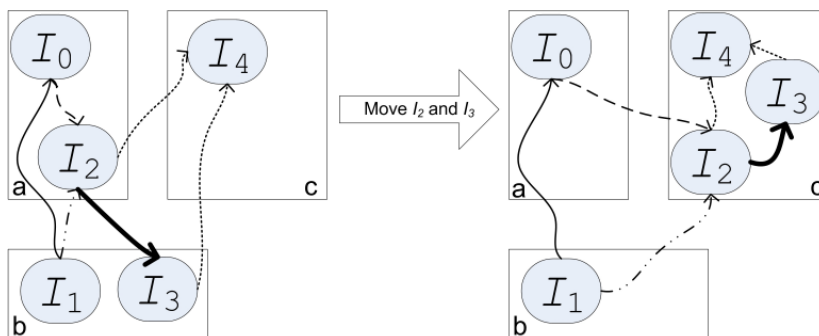


Figure 5.2: Moving tasks eliminates edges and exposes new ones


```

1: optimizeBacktrack( sol, CP,  $\mathcal{J}$ ,  $\mathcal{C}$  )
2: for each non-mapped edge, compute  $t = L_k D_{i_1 i_2} Z_{i_1 i_2} + \tau_k p_k B_{i_3, i_4} z'_k$  and store in list
3: last =  $\infty$ 
4: if makespan mode then
5:   current = longestPathValue( $\mathcal{G}$ ,  $\mathcal{J}$ ,  $\mathcal{C}$ )
6: else
7:    $current = \sum_{i=0}^n \sum_{j=0}^m x_{ij} \cdot T_{ij} + T_{switch} + T_{net}$ 
8: end if
9: while current  $\leq$  last AND (list is not empty) AND (number of trials is not reached) do
10:  remove the largest element in list
11:  if corresponding edge  $I_{i_1} < I_{i_2}$  is not fixed then
12:    deassign  $I_{i_1} < I_{i_2}$ 
13:    deassign  $I_{i_1}$  and  $I_{i_2}$ 
14:     $G_l = \{I_{i_1}, I_{i_2}\}$ 
15:    allocateProcessorForGroup( $G_l, \mathcal{J}, CP$ )
16:    if  $I_{i_1}$  or  $I_{i_2}$  is not mapped then
17:      undo edge and task deassignment
18:    else
19:      if there is an edge incident on  $I_{i_1}$  or  $I_{i_2}$  to tasks on previous processor then
20:        assign those edges
21:        if failed to assign OR new solution is more expensive then
22:          undo edge and task deassignment for  $I_{i_1}$  and  $I_{i_2}$ 
23:        else
24:          add the new edges to list
25:        end if
26:      end if
27:      if condition 19 was FALSE OR condition 21 was FALSE then
28:        deassign all edges incident on the same processor and remove them from list
29:        last = current
30:        if makespan mode then
31:          current = longestPathValue( $\mathcal{G}$ ,  $\mathcal{J}$ ,  $\mathcal{C}$ )
32:        else
33:           $current = \sum_{i=0}^n \sum_{j=0}^m x_{ij} \cdot T_{ij} + T_{switch} + T_{net}$ 
34:        end if
35:      end if
36:    end if
37:  end if
38: end while
39: return sol

```

Algorithm 12: Optimization loop using backtracking

- New edges can be assigned, but the resulting solution is more expensive.

In either case, the algorithm backtracks. Otherwise, newly exposed and assigned edges ($I_0 < I_2$ in the example) are added to the list to be considered for elimination in following iterations. This addition is the reason behind the third termination condition for the main loop: obviously, we can have a situation where one iteration eliminates an edge exposing a new one, and the next iteration reverses the process. The third termination safeguards indefinite loops. In this implementation, the maximum number of trials is arbitrarily selected to equal the total number of edges in the application graph G . Backtracking and addition of edges to the list means that the optimization loop is generally not greedy, exceptions being cases in which the heuristic does not take these paths.

If assigning newly exposed edges is successful, the next step is to deassign edges that were previously assigned, but now no longer cross processor boundaries because of task movements, and to remove them from the list containing edges to be considered for elimination (in the example, the edges $I_2 < I_4$ and $I_3 < I_4$). This is done from Line 27, where also the value of the objective for the new solution is computed. Note that we do not consider what happens to the edge $I_1 < I_2$ in the example which is now mapped to a network resource which connects between another set of processors. The reason is that the communication cost does not change. What changes is the cost through bridges (see Figure 3.2). While this overhead cost is assumed to be zero in this implementation, the heuristic can be easily extended to include the cost.

5.2.4 Area Recovery

The optimization loop operates without any regard to FPGA area. Therefore, synthesis is completed by Algorithm 13 which attempts to fit the optimized solution through resource sharing and deallocation of resources. Still working under the assumption that inter-task communications are more expensive, area optimization begins by deallocating processors in order to have a minimum impact on the overall solution (Lines 2-21). The processor deallocation loop is succeeded by a network deallocation loop if the architecture does not meet network area constraint.

The first loop deallocates processors in order of increasing cost, which is the sum of execution and scheduling costs of tasks mapped on processors. Doing so increases the cost for affected tasks, and increases scheduling cost for other tasks which now have to share their processor. Thus, starting with lowest cost processors minimizes the overall impact. However, deallocation is done only if the processor contains no fixed tasks when synthesizing in makespan mode.

Each deallocation is followed by an attempt to individually assign each of the affected tasks. If assignment fails, the problem is declared infeasible. Mapping the tasks individually reduces the chance that this process fails.

All edges incident on affected tasks are deassigned, and those which still cross processor boundaries after task movements are assigned. Because no network has been deallocated at this stage, each new assignment ends successfully. Moreover, no check for fixed edges is required at this step prior to deassigning, because the tasks to which the edge is incident are not fixed.

Note that deallocation and reassignment proceed in a greedy-fashion, but that the problem has resource constraints so that the iterative deallocation may lead to less critical tasks blocking better resources which would otherwise be occupied by tasks deassigned through subsequent deallocations.

The second loop for deallocating networks proceeds similarly. As in the first loop, the call for `netAssign()` additionally leads to task deassignment in order to map affected tasks on the same processor.

```

1: satisfyArea( $G, A_{PE}, A_{net}, \mathcal{J}, \mathcal{C}, CP$ )
2: while  $\sum_{j=0, J_j \text{ allocated}}^m a_j \geq A_{PE}$  AND (not all processors have been iterated through) do
3:   find  $J_j$  with the lowest sum of execution and scheduling cost
4:   if  $J_j$  contains non-fixed task then
5:     deallocate  $J_j$  so that  $G_l$  previously on  $J_j$  becomes unmapped
6:     deassign all edges incident on each task  $I_i \in G_l$  and put them in a list
7:     for each  $I_i \in G_l$  do
8:       allocateProcessorForGroup( $I_i, \mathcal{J}, CP$ )
9:       if  $I_i$  is not mapped then
10:        report problem not feasible
11:       end if
12:     end for
13:     sort list in order of descending costs
14:     for each edge  $I_{i_1} < I_{i_2}$  in list do
15:       if  $I_{i_1}$  and  $I_{i_2}$  are on different processors then
16:         netAssign( $I_{i_1} < I_{i_2}, \mathcal{C}, list, CP$ )
17:       end if
18:     end for
19:     remove  $J_j$  from solution set
20:   end if
21: end while
22: if  $\sum_{j=0, J_j \text{ allocated}}^m a_j \geq A_{PE}$  then
23:   report problem not feasible
24: end if
25: while  $\sum_{k=0, \text{ allocated}}^K A_k \geq A_{net}$  AND (not all networks have been iterated through) do
26:   find  $y_k$  with the lowest sum of latency and arbitration cost
27:   if  $y_k$  contains no fixed edge then
28:     deassign all edges mapped on  $y_k$  and put them in list
29:     sort list in descending order
30:     repeat steps 14-18 for list
31:     remove  $y_k$  from solution set
32:   end if
33: end while
34: if  $\sum_{k=0, \text{ allocated}}^K A_k \geq A_{net}$  then
35:   report problem not feasible
36: end if
37: return sol

```

Algorithm 13: Area optimization

5.3 Group Growing Heuristic

Restarting a search rather than backtracking and tracing a different path is an alternative strategy that can potentially undo bad decisions committed at earlier stages because the heuristic presented in the previous section only traces one step back in search of better solutions.

The difference between the heuristic presented in this section and the previous one is in the optimization loop which is shown in Algorithm 14. As before, if tasks affected by deassigning an edge are successfully mapped on another processor, an attempt is made to assign newly exposed edges incident on those tasks, going to tasks on previous processors (edge $I_0 < I_2$ in example of Figure 5.2). If the edge assignment is successful but the solution is more expensive, then most likely the reason is additional heavy traffic due to the newly exposed edge. Therefore, the heuristic searches in Line 24 for the most expensive non-fixed edge incident on a task in the newly mapped group G_l , whose other end is incident on a task I_{i^*} not in G_l . When found, the edge is deassigned, and the task not in G_l is added to the group. An attempt is then made to map the new group so that tasks with expensive edges between them are mapped on the same processor as exemplified in Figure 5.3.

Effectively, the heuristic jumps back to a step where the task I_{i^*} is not yet mapped, and restarts the search from that point. The growing of the group G_l to encompass expensive edges stops if the group becomes infeasible to map, or if newly exposed edges do not result into a more expensive solution.

5.4 Priority Assignment Heuristic

This heuristic is implemented to compare how a communication-centric approach as discussed in the previous two subsections performs relative to a method that makes no assumptions about the dominance of inter-task communication cost over task execution and scheduling overhead.

The heuristic outlined in Algorithm 15 does not begin with an hypothetical solution, but rather, attempts to map tasks and edges solely in order of monotonically decreasing cost.

The hypothetical assignment in Line 1 is only used to determine the order of the weights. The main loop composes a solution from scratch. At any step, assignment is based on what is the most expensive weight. If the weight is for task execution cost, the assignment attempt between Lines 7-11 is straight forward: the task is assigned, or if the attempt fails, the problem is declared infeasible. If on the other hand the most-expensive weight corresponds to an edge, several checks are necessary to insure that a solution is valid (corresponding to Line 5 in Algorithm 5) to check if extending would maintain hereditary property in case of problems without resource constraints.

If the two tasks on which the edge is incident have been mapped in a previous step, and are on different processors, then a network is allocated for the edge. A failure implies that a feasible solution was not found.

If only one of the task is mapped, then rather than allocating a network for the edge, an attempt is made to map the other task on the same processor to minimize the cost. In case that is not possible, another processor is allocated, followed by a network allocation for the edge.

Otherwise, if none of the two tasks are mapped, an attempt is made to map the two as a group, again, to minimize the cost of inter-task communications. If the attempt fails, then the two tasks are assigned independently, followed by the edge assignment when successful.

The heuristic is completed with an area recovery step which is the same as in the other two heuristics. When synthesizing using this method, `priorityAssign()` sits the heart of the makespan synthesis loop, rather than the main Algorithm 6.

```

1: optimizeGrow( sol, CP,  $\mathcal{J}$ ,  $\mathcal{C}$ )
2: for each non-mapped edge, compute  $t = L_k D_{i_1 i_2} Z_{i_1 i_2} + \tau_k p_k B_{i_3, i_4} z'_k$  and store in list
3: last =  $\infty$ 
4: if makespan mode then
5:   current = longestPathValue( $\mathcal{G}$ ,  $\mathcal{J}$ ,  $\mathcal{C}$ )
6: else
7:    $current = \sum_{i=0}^n \sum_{j=0}^m x_{ij} \cdot T_{ij} + T_{switch} + T_{net}$ 
8: end if
9: while current  $\leq$  last AND (list is not empty) AND (number of trials is not reached) do
10:  remove the largest element in list
11:  if corresponding edge  $I_{i_1} < I_{i_2}$  is not fixed then
12:    deassign  $I_{i_1} < I_{i_2}$ 
13:    deassign  $I_{i_1}$  and  $I_{i_2}$ 
14:     $G_l = \{I_{i_1}, I_{i_2}\}$ 
15:    repeat
16:      allocateProcessorForGroup( $G_l, \mathcal{J}, CP$ )
17:      if  $G_l$  is not mapped then
18:        undo edge and task deassignment
19:      else
20:        if there is an edge incident on any  $I_i \in G_l$  to tasks on previous processor then
21:          assign those new edges
22:          if new solution is more expensive then
23:             $G'_l = \{\mathcal{T} \setminus G_l\}$ 
24:            find non-fixed  $I_{i^*} \in G'_l \mid \forall I_i \in G_l, I_{i_1} \in G'_l, \text{cost}(I_{i^*} < I_i) \geq \text{cost}(I_{i_1} < I_i)$ 
25:             $G_l = G_l \cup I_{i^*}$ 
26:            deassign  $I_{i^*}$ 
27:          else if failed to assign new edges then
28:            undo edge and task deassignment for each  $I_i \in G_l$ 
29:          else
30:            add the new edges to list
31:          end if
32:        end if
33:        if condition 20 was FALSE OR condition 27 was FALSE then
34:          deassign all edges incident on the same processor and remove them from list
35:          last = current
36:          compute current according to 4-8
37:        end if
38:      end if
39:    until  $G_l$  fails to map OR new edges are assigned
40:  end if
41: end while
42: return sol

```

Algorithm 14: Optimizing loop using group growing

```

1: priorityAssign(  $G, \mathcal{J}, \mathcal{C}, CP, A_{PE}, A_{net}$  )
2:  $sol = \text{hypotheticalAssignment}(G, \mathcal{J}, \mathcal{C})$ 
3:  $nCost = \text{sortNodeCosts}(sol)$ 
4:  $eCost = \text{sortEdgeCosts}(sol)$ 
5: while  $nCost$  AND  $eCost$  have elements left do
6:    $cost = \max(\text{nextElement}(nCost), \text{nextElement}(eCost))$ 
7:   if  $cost$  corresponds to cost of a non-assigned task  $I_i$  then
8:      $\text{allocateProcessorForGroup}(\{I_i\}, \mathcal{J}, CP)$ 
9:     if  $I_i$  is not assigned then
10:       report problem not feasible
11:     end if
12:   else if  $cost$  corresponds to non-assigned edge  $I_{i_1} < I_{i_2}$  then
13:     if  $I_{i_1}$  AND  $I_{i_2}$  are assigned AND are on different processors then
14:        $\text{allocateNetwork}(I_{i_1} < I_{i_2}, \mathcal{C})$ 
15:       if  $I_{i_1} < I_{i_2}$  is not assigned then
16:         report problem not feasible
17:       end if
18:     else if if only one of  $I_{i_1}, I_{i_2}$  is assigned to a  $J_j$  then
19:       try to map the non-assigned task to  $J_j$ 
20:       if assignment fail then
21:          $\text{allocateProcessorForGroup}(\{\text{the task}\}, \mathcal{J}, CP)$ 
22:         if assignment fails then
23:           report problem not feasible
24:         else
25:            $\text{allocateNetwork}(I_{i_1} < I_{i_2}, \mathcal{C})$ 
26:           if  $I_{i_1} < I_{i_2}$  is not assigned then
27:             report problem not feasible
28:           end if
29:         end if
30:       end if
31:     else
32:        $g = \{I_{i_1}, I_{i_2}\}$ 
33:        $\text{allocateProcessorForGroup}(g, \mathcal{J}, CP)$ 
34:       if assignment fails then
35:         try to assign  $I_{i_1}$  and  $I_{i_2}$  independently
36:         if assignment for either  $I_{i_1}$  or  $I_{i_2}$  fails then
37:           report problem not feasible
38:         else
39:            $\text{allocateNetwork}(I_{i_1} < I_{i_2}, \mathcal{C})$ 
40:           if  $I_{i_1} < I_{i_2}$  is not assigned then
41:             report problem not feasible
42:           end if
43:         end if
44:       end if
45:     end if
46:   end if
47: end while
48:  $sol = \text{satisfyArea}(sol, A_{PE}, A_{net}, \mathcal{J}, \mathcal{C}, CP)$ 
49: return  $sol$ 

```

Algorithm 15: Priority based assignment

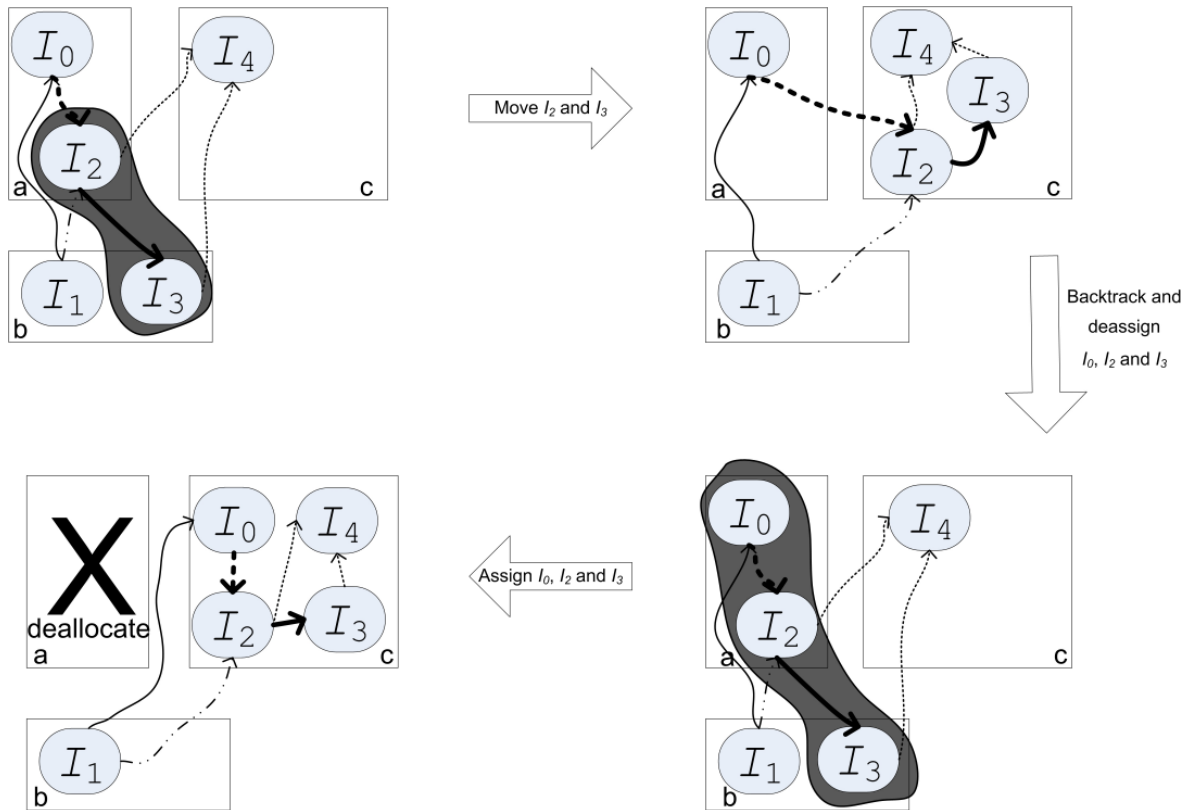


Figure 5.3: Group growing

5.5 Comparison of Synthesis Results

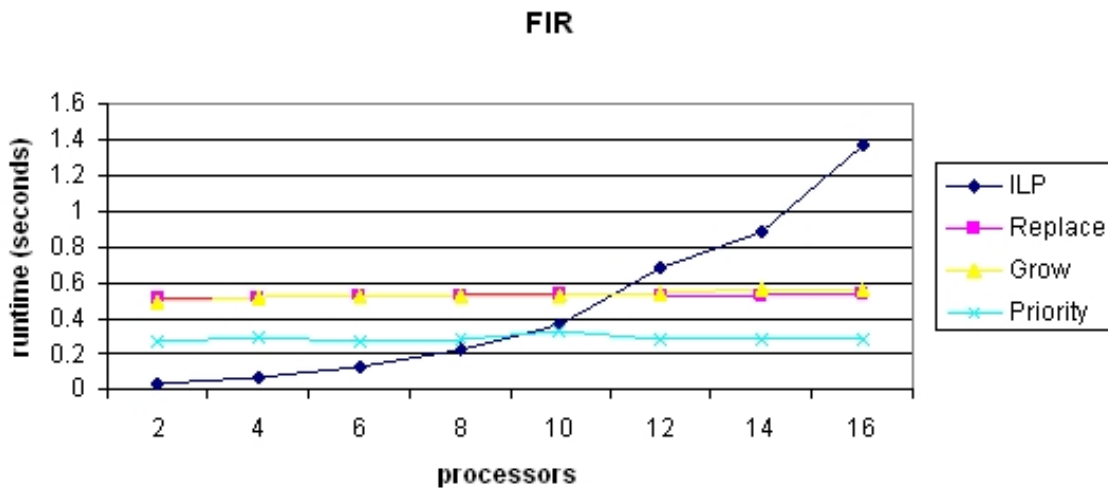
This section presents a comparison of synthesis results for the three heuristics against ILP-based synthesis both in terms of runtime and the quality of the solutions obtained. Similarly to Section 4.4, comparison was conducted for the six non-realtime applications, preemptive scheduling and makespan optimization. The flow of Figure 3.1 was extended once again to support synthesis using these heuristics. The same set of processors and networks were used.

5.5.1 Non-Realtime Applications

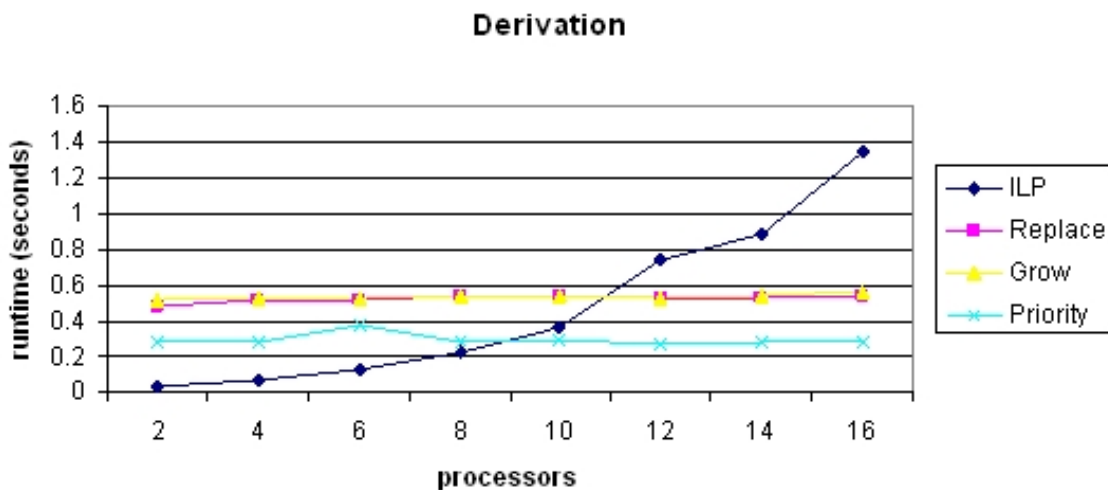
Figure 5.4 compares the runtime when the number of processors is increased from 2 to 16, while that of tasks and networks is kept at 4 and 2 respectively. As expected, all three heuristics have a nearly constant time because there is no extensive search. Moreover, all three found the same optimum solution as ILP-based synthesis¹. The latter result does not follow from the fact that problem scenarios in this case do not have resource constraints, but rather, because the right decisions were made from the beginning. Since the heuristics are not greedy even when problems have no resource constraints, bad decisions committed at the beginning could lead to suboptimum results.

Comparing the heuristics against each other, priority-based assignment is much faster because this is essentially a single-pass resolution as opposed to the other two. The other two have an almost the same performance because there is no much opportunity to grow groups of tasks. One would otherwise expect replace-heuristic to perform better from a runtime point of view, the reason being that searching is limited to single-step backtracking.

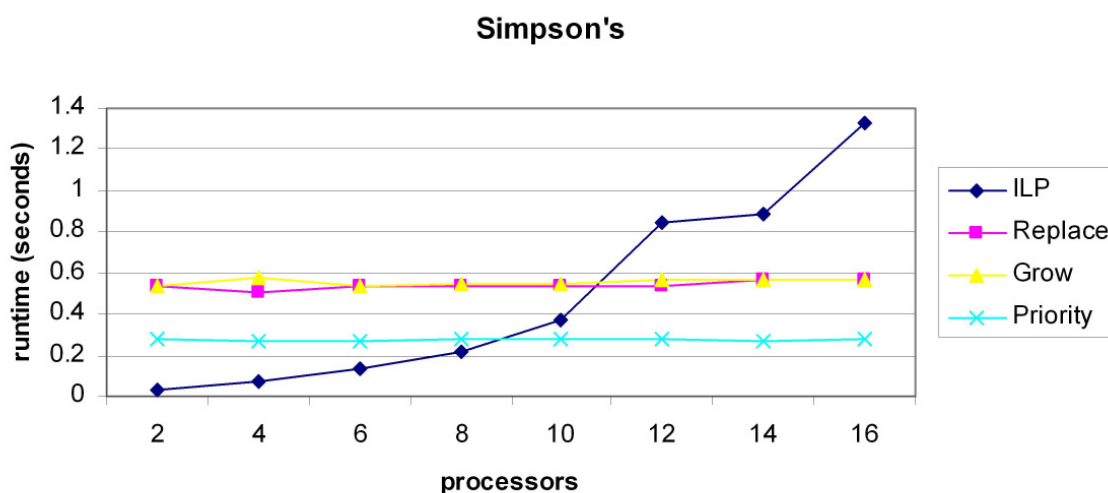
¹Value of objectives in seconds for FIR, Derivation, Simpsons, N-Body and Inversion respectively: 1.88, 0.42, 0.24, 0.02, 1.58 and 43.44.



(a) FIR

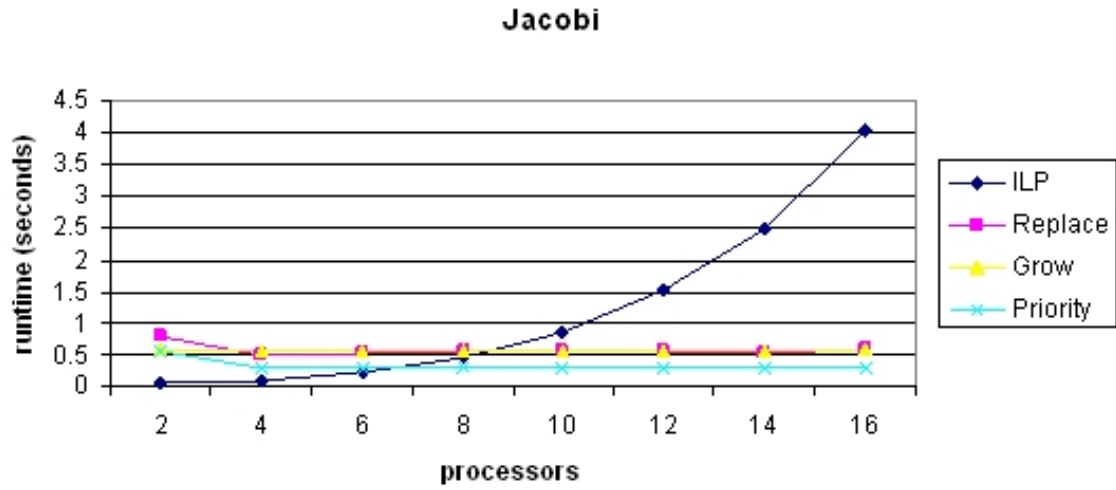


(b) Derivation

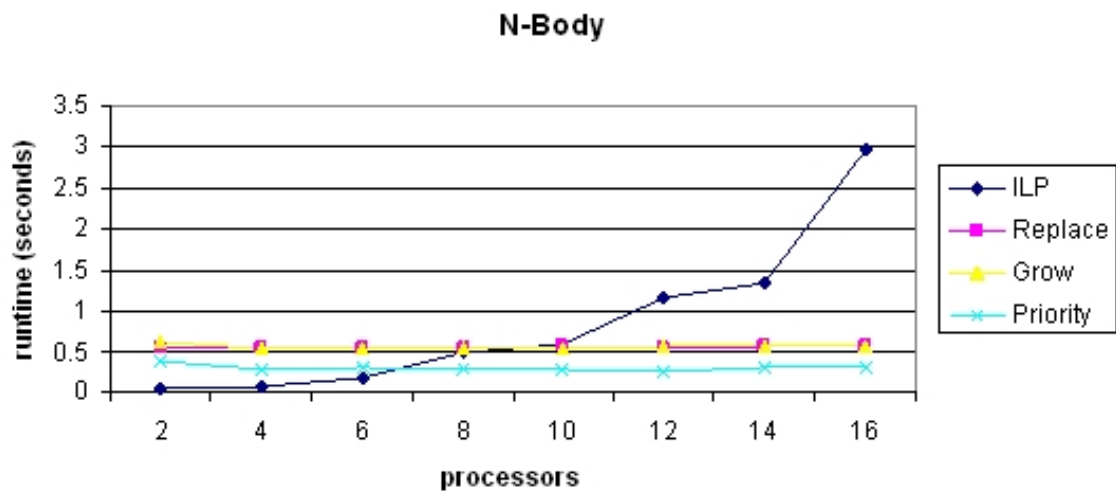


(c) Simpsons

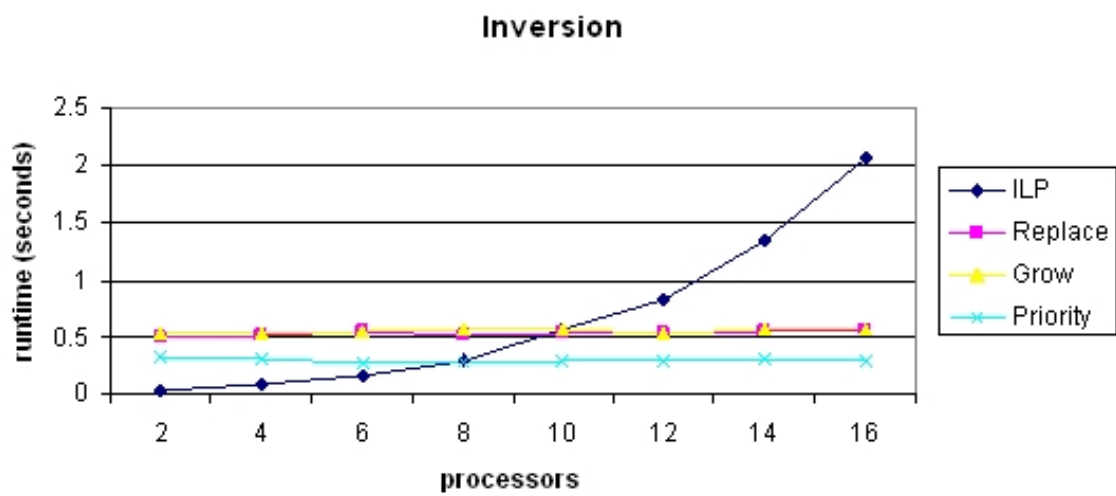
Figure 5.4: Continued on next page



(a) Jacobi



(b) N-Body



(c) Inversion

Figure 5.4: Greedy-ILP comparison : Increasing the number of processors

Figure 5.5 shows the comparison when the number of networks is progressively increased to 5. This figure contains no surprises based on prior observations in this section and in previous two chapters: nearly flat runtime, heuristics are faster, optimum solutions in each case, priority-based assignment is the fastest.

As before, increasing the number of tasks is much more revealing. Figure 5.6 shows what happens when the number of tasks is now increased to 22, while keeping that of processors and networks at 16 and 5 respectively. The trends for ILP are the same as in Figures 3.8 and 4.5: there is a timeout at 28800 seconds with suboptimum solutions for points at that line, failure to find solutions for points above the line, and otherwise optimum solutions. Not surprising, the heuristics are significantly faster by several orders of magnitude, where priority-based assignment is the fastest.

Interesting is that, in contrast to the other two heuristics, priority-based assignments appears to generally fail to find feasible solutions when the number of tasks becomes larger. This is a direct consequence of initial bad decisions: in presence of resource constraints, assignment based on cost can block resources needed by other tasks and edges. If there is not backtracking or restarts, heuristics may fail to find solutions. The implication is that, if we were to implement a greedy synthesis algorithm according to Algorithm 5 by doing a full schedulability analysis a priori, but the problem is resource constrained, we can expect our algorithm to fail to find a solution when the problem is sufficiently complex.

Comparing the other two heuristics against each other, we note as expected that group growing tends to take a longer time to finish because of restarts. Otherwise, both exhibit a moderate increase in runtime with larger number of tasks, which is very promising for automated synthesis provided the quality of results is acceptable.

Figure 5.7 shows the value of the objectives obtained with each approach. What is observed is that there is no heuristic that is generally superior. With the exception of synthesis for Inversion, there was always a heuristic which delivered results close to ILP-based synthesis. Therefore, given that synthesis time for each heuristic is very short, a reasonable strategy for obtaining good architectures would be a voting scheme which employ all three heuristics followed by selecting the best architecture.

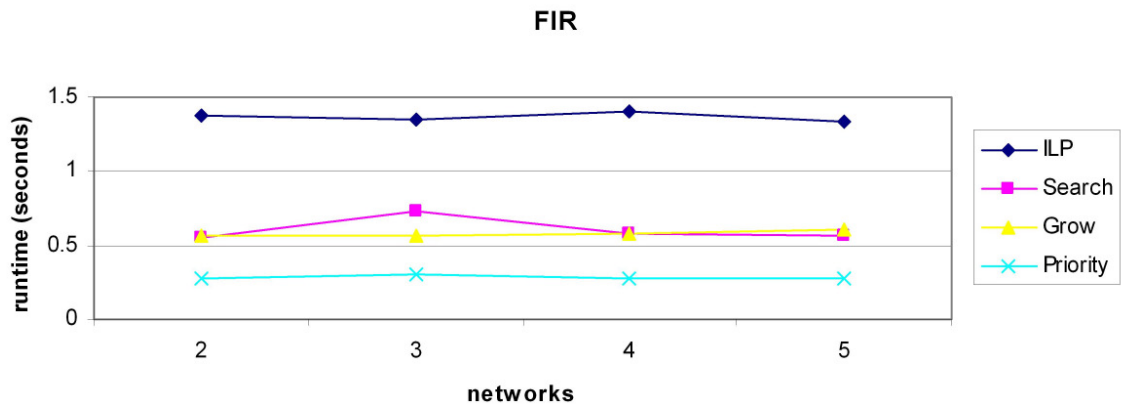
5.5.2 Realtime Preemptive Scheduling

Table 5.1 summarizes the comparison of synthesis results obtained using the three heuristics against ILP under realtime preemptive scheduling. As in Section 3.2.2.6, we additionally compare the synthesis run time with and without fixed-priority preemptive scheduling using configurations with a basic cyclic executive and with a preemptive kernel.

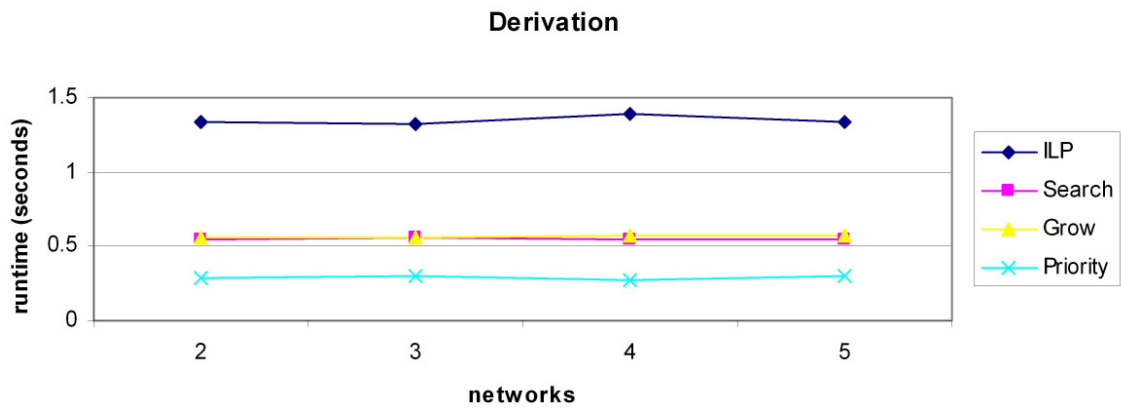
The general picture obtained in this table is the same as the one in the foregoing section: all heuristics are faster by several orders of magnitude, and priority-based assignment is the fastest of all. Moreover, it is again noticeable that with the exception of one case for WCDMA under cyclic executive, at least one heuristic returned a result close to the optimum. This result is in favor of the voting scheme proposed in the previous subsection.

5.5.3 Makespan Optimization

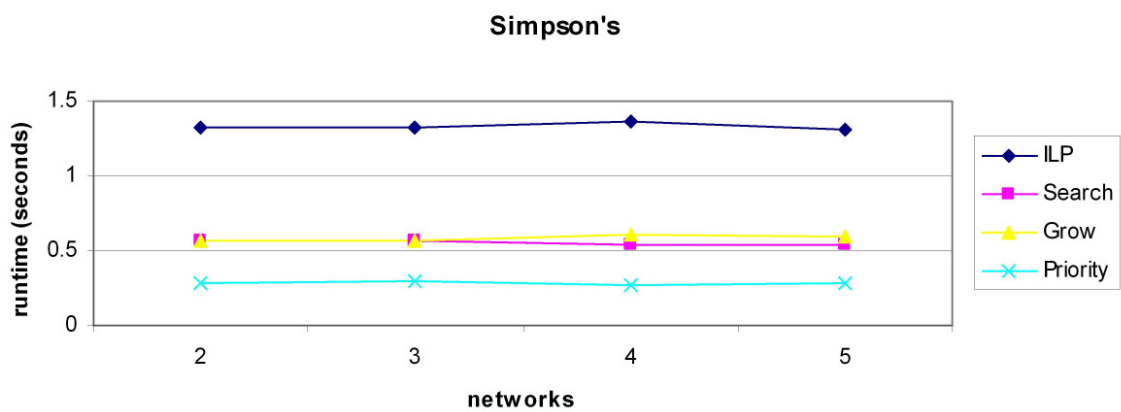
Finally, Table 5.2 completes the comparison under makespan optimization. Even though as expected the heuristics are very fast, the computation of the longest path was wrong leading to incorrect decisions. This aspect of makespan optimization is an open problem that needs to be addressed. However, given the results in the two previous subsections, it is expected that a voting scheme using all heuristics would be an effective strategy for makespan optimizations.



(a) FIR

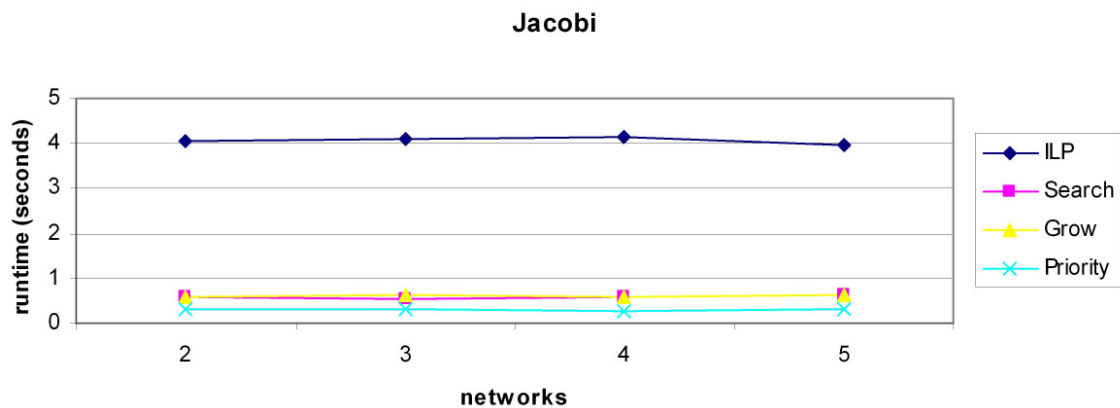


(b) Derivation

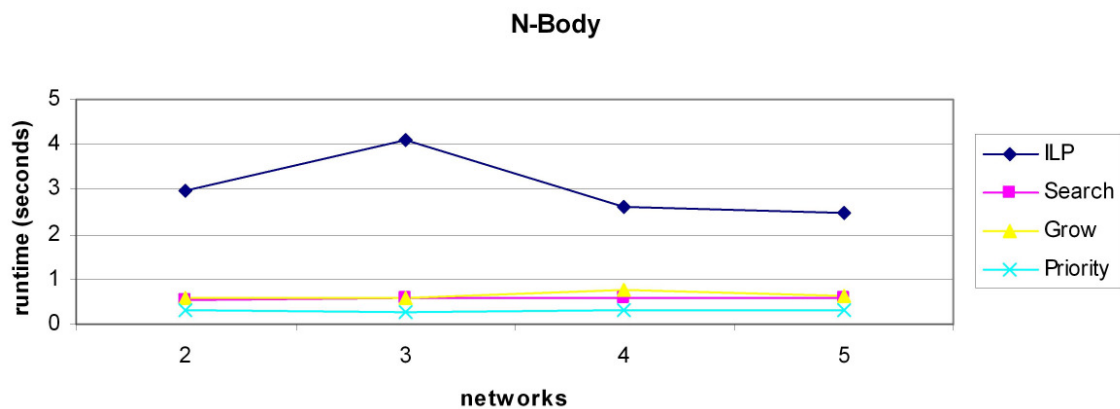


(c) Simpsons

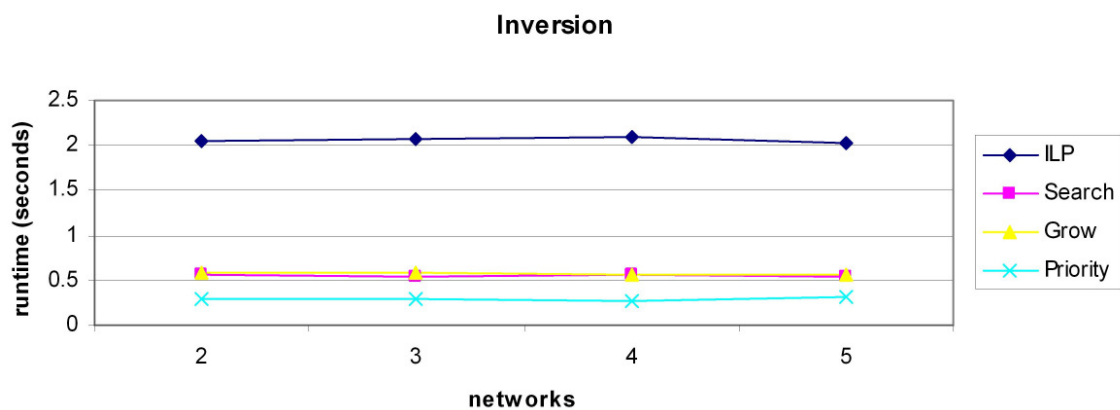
Figure 5.5: Continued on next page



(a) Jacobi

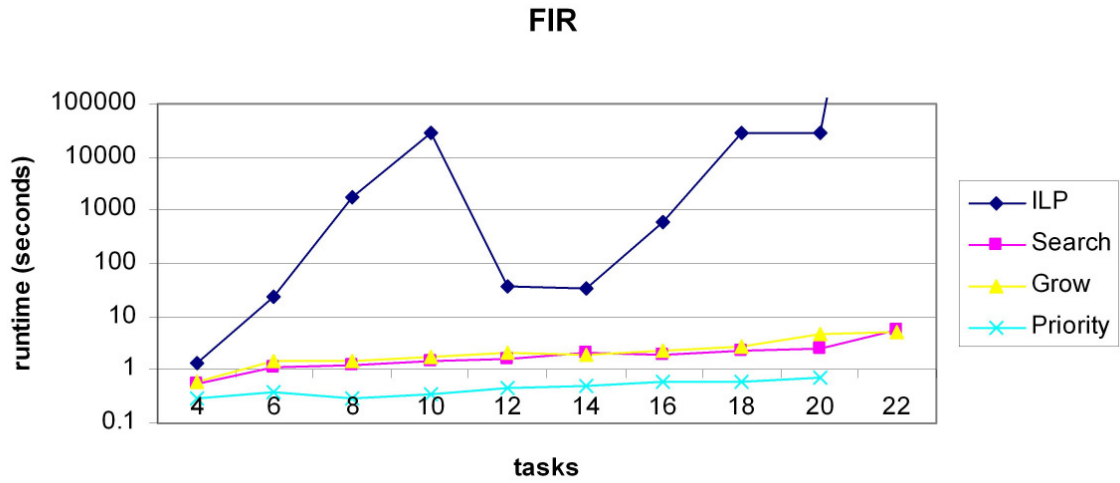


(b) N-Body

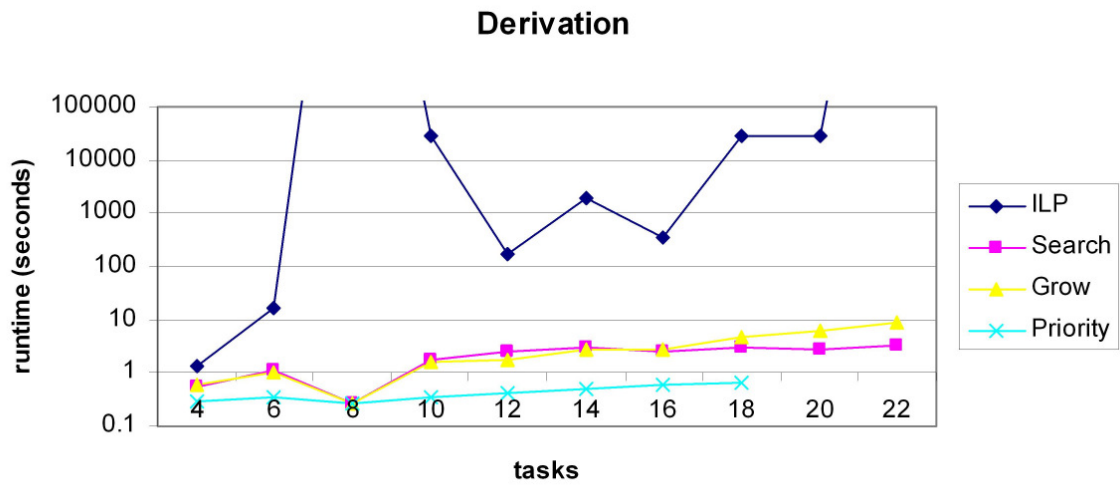


(c) Inversion

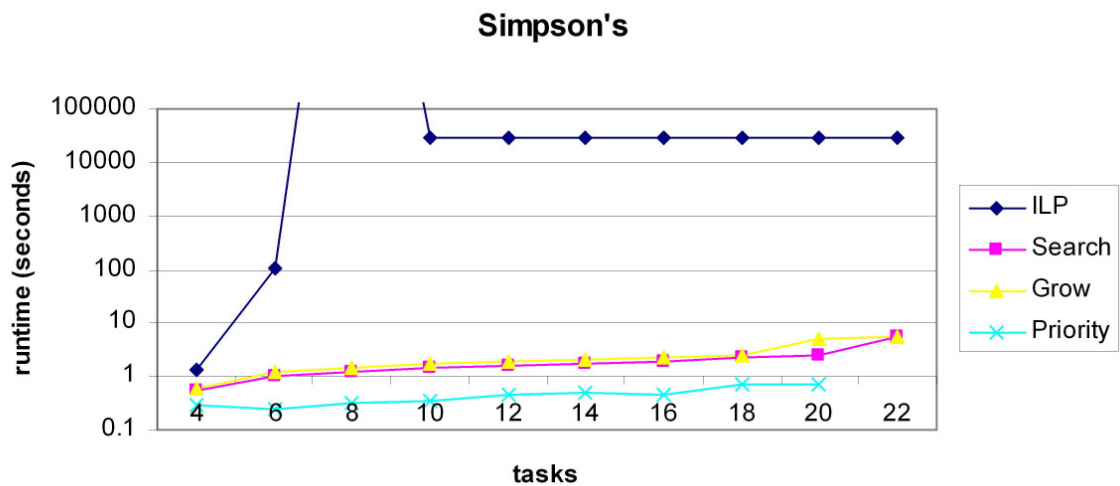
Figure 5.5: Greedy-ILP comparison : Increasing the number of networks



(a) FIR:Priority failed from $n \geq 22$

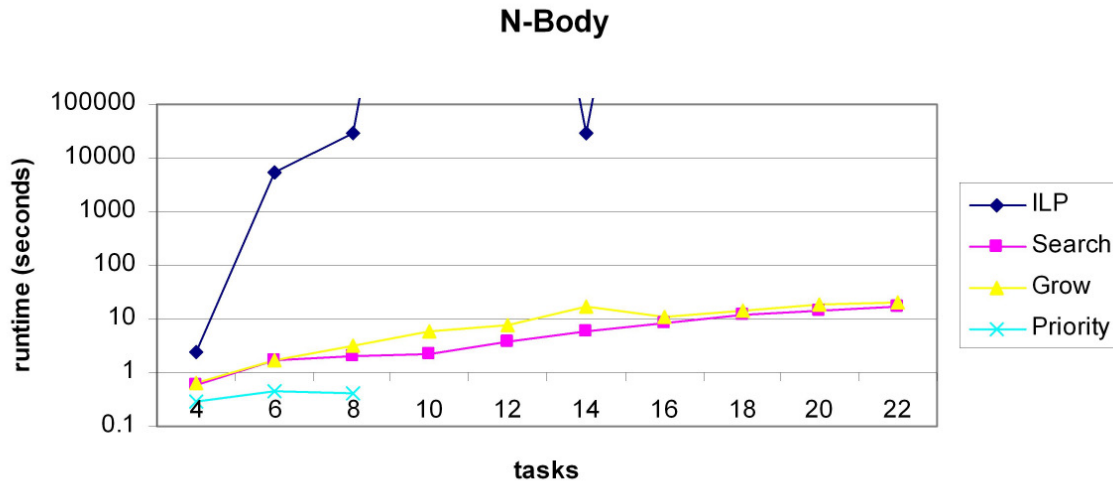


(b) Derivation:Priority failed from $n \geq 20$

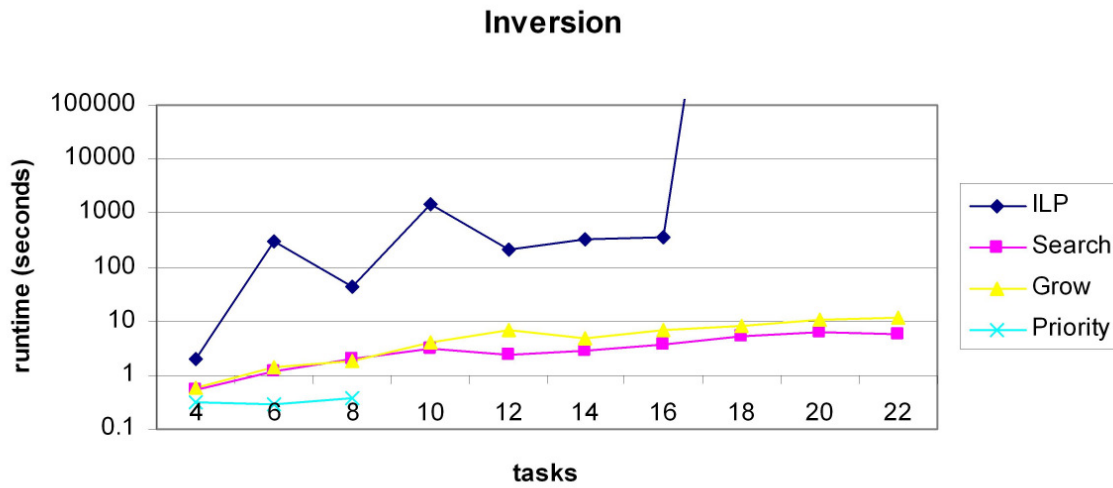


(c) Simpsons:Priority failed from $n \geq 22$

Figure 5.6: Continued on next page



(a) N-Body:Priority failed from $n \geq 10$

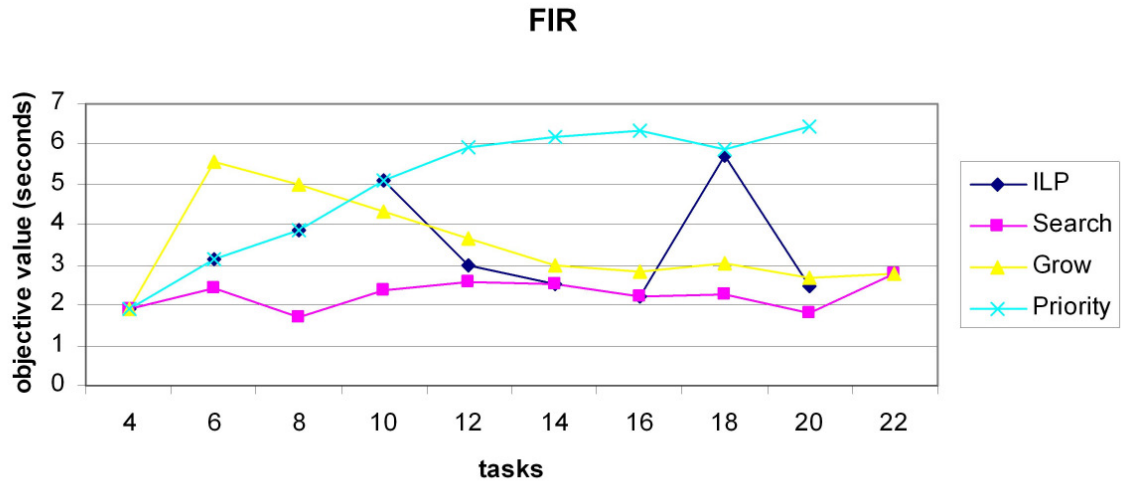


(b) Inversion:Priority failed from $n \geq 10$

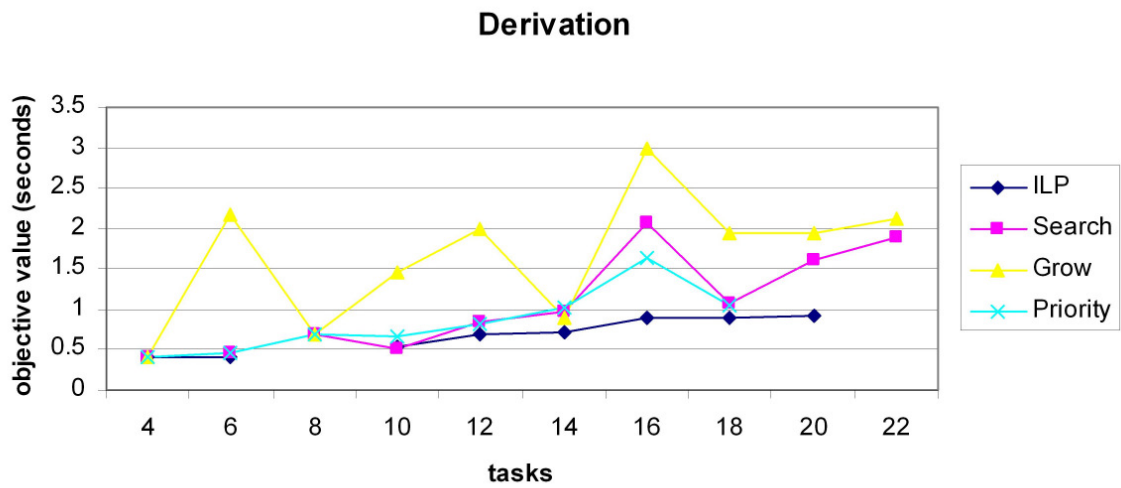
Figure 5.6: Greedy-ILP comparison : Increasing the number of tasks

Table 5.1: Greedy-ILP comparison under scheduling

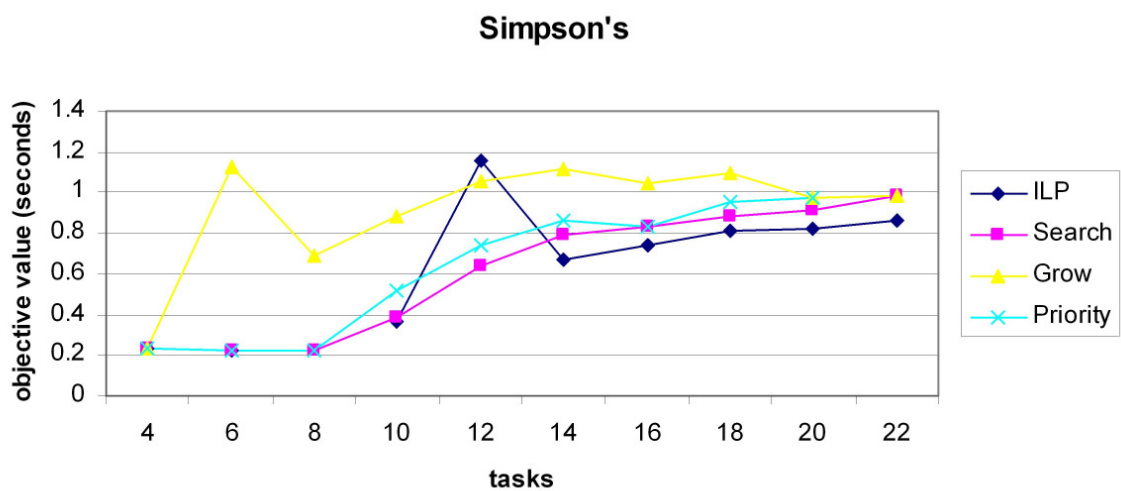
| Alg. | Appl. | Cyclic | | Preemptive | |
|----------|-------|----------------|------------|----------------|------------|
| | | Run time (sec) | Obj.(sec.) | Run time (sec) | Obj.(sec.) |
| ILP | WLAN | 636 | 0.0001 | 1010 | 0.10 |
| | WCDMA | 28811 | 18.25 | 19567 | 16.90 |
| Search | WLAN | 2.313 | 14.7972 | 4.344 | 14.7543 |
| | WCDMA | 1.906 | 64.3659 | 8.047 | 18.6025 |
| Grow | WLAN | 5.500 | 0.0002 | 5.656 | 0.0852 |
| | WCDMA | 2.062 | 62.0416 | 8.687 | 18.2749 |
| Priority | WLAN | 0.984 | 0.1349 | 1.656 | 14.7543 |
| | WCDMA | 1.016 | 62.8219 | 2.141 | 19.2913 |



(a) FIR:Priority failed from $n \geq 22$

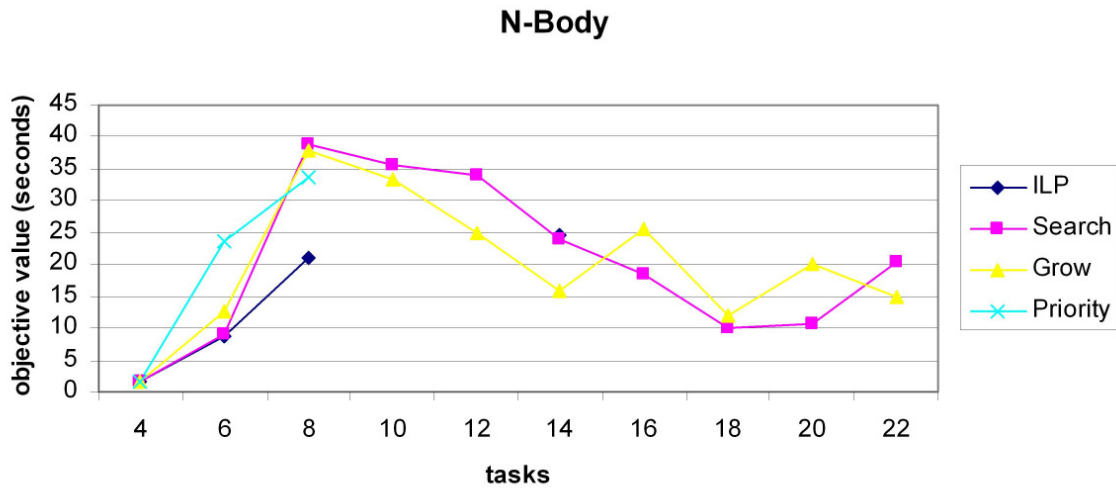


(b) Derivation:Priority failed from $n \geq 20$

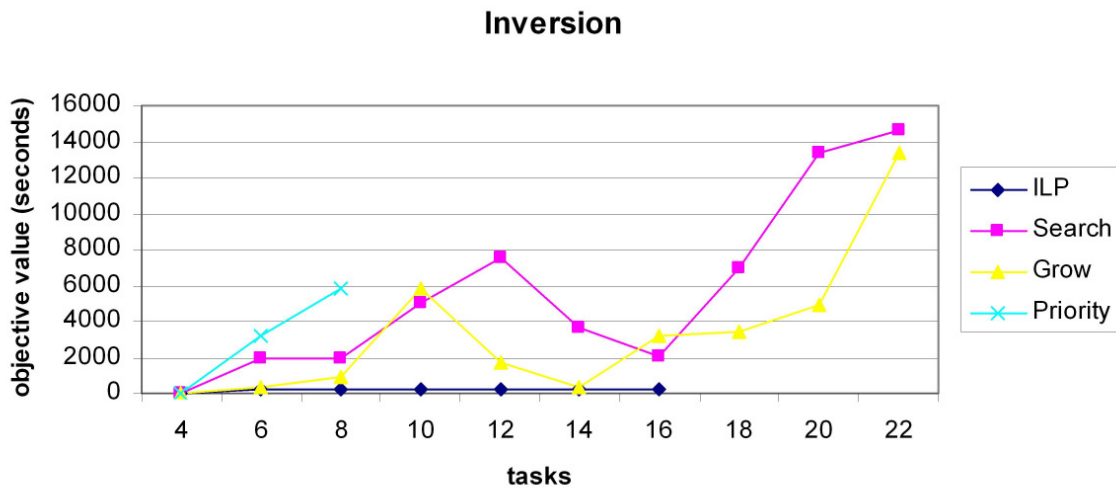


(c) Simpsons:Priority failed from $n \geq 22$

Figure 5.7: Continued on next page



(a) N-Body:Priority failed from $n \geq 10$



(b) Inversion:Priority failed from $n \geq 10$

Figure 5.7: Greedy-ILP Comparison : Increasing the number of tasks, objective value

Table 5.2: Greedy-ILP comparison for makespan optimization

| Alg. | Cyclic | | | Preemptive | | |
|----------|----------------------------|----------------|------------|----------------------------|----------------|------------|
| | Final CP | Run Time (sec) | Obj.(sec.) | Final CP | Run Time (sec) | Obj.(sec.) |
| ILP | I0 I7 I6 I5 I4 I2 I1 I3 | 808.688 | 32.9932 | I0 I7 I6 I5 I4 I1 I3 | 5020.72 | 32.9932 |
| Search | I0 I1 I3 | 5.758 | 0.0485 | I0 I1 I3 | 18.484 | 0.0019 |
| Grow | I0 I1 I3 | 5.282 | 0.03829 | I0 I1 I3 | 18.484 | 0.0019 |
| Priority | I0 I1 I3 | 1.899 | 1.782 | I0 I7 I6 I5 I4 I2 I1 I3 | 0.0136 | 14.7608 |

5.6 Chapter Summary

This chapter presented three heuristics with the intent of enabling fast synthesis for large problems. It was shown that there exist an optimum linear time greedy algorithm for problem instances without resource constraints. However, a purely greedy algorithm likely requires a full schedulability analysis, with similar drawbacks as those faced in ASP-based synthesis. The heuristics introduced avoid a full analysis by assuming that scheduling overhead can be ignored.

In addition to the assumption above, two of the heuristics exploit an observation made in makespan optimization in Chapter 3 by assigning best resources for each node and edge, followed by an optimization loop based on resource sharing. The latter is a local search heuristic, where backtracking and restarting are attempted respectively.

Synthesis results showed that all three heuristics are very fast. Further, it was apparent that none of the heuristic is overall superior in terms of the quality of synthesized architectures. However, in most synthesis scenarios conducted, at least one of the heuristic delivered results close to or equal to those obtained using ILP-based synthesis. Since synthesis time is very fast, it appears that a good synthesis strategy is to employ a voting scheme, where all three heuristics are used, and the best result is selected.

6 Conclusion

6.1 Summary

In this thesis, we investigated automated synthesis of flexible chip multiprocessor systems targeted for reconfigurable devices such as FPGAs. Flexible CMP systems address power and memory wall challenges by combining advantages from multiprocessing and reconfigurability to exploit both task-level parallelism and system customization. We showed that the design space of these systems is very large so that disciplined work-bench based explorations are infeasible, making automatic synthesis necessary. We used the detailed description of the design space to determine which parameters should be modeled to facilitate automated synthesis by optimizing a cost function. In the process, emphasis was placed on inclusive modeling of parameters from application, architectural and physical subspaces, as well as their joint coverage in order to avoid pre-constraining the design space, an aspect that generally leads to suboptimum architectures. It was shown that inclusiveness and joint coverage is necessitated by cross-effects between design space dimensions, and between the three subspaces. Therefore, the high-level automated synthesis problem consists of simultaneous activities of (i) selecting processors (ii) mapping and scheduling tasks to them, and (iii) selecting one or several networks for inter-task communications such that design constraints and optimization objectives are met.

To enable automatic synthesis, we developed an ILP model for the problem, and proposed a design flow that takes parallel programs as input to produce a high-level netlist and information on which communication resources a task should use to communicate with any other task. Because the model is not pre-constraining, multiple communication resources can be allocated between any pair of processors to minimize expensive inter-task communications, an aspect which is in stark contrast to related work.

We evaluated the model using non-realtime and realtime applications under preemptive scheduling, and in the process showed that automated synthesis is absolutely important because the combinatorial nature of the problem can lead to counter intuitive results so that skilled designers can easily miss superior design points when conducting manual explorations. Moreover, we showed how designers can reduce synthesis runtime by comparing task execution time and traffic pattern to eliminate number of processors from solution space without limiting the design space. Furthermore, we investigated synthesis via answer set programming and by using application-specific heuristics to address runtime challenges for large problem instances.

Synthesis through ASP was investigated to determine how well state of the art techniques from the domain of boolean satisfiability can be exploited to solve difficult problem instances. Using the fact that 0-1 ILP problems can be encoded into equivalent ASP programs, we used the ASP solver clasp to show that such techniques are indeed very effective, where the synthesis runtime could be reduced by up to three orders of magnitude. Not all problems could be solved through ASP, the reason being that both ILP and ASP-based synthesis currently require a full schedulability analysis for all possible task mappings on all processors, thereby leading to an explosion of decision variables, linear constraints, atoms and rules. We determined that where ASP-based synthesis failed, the solver did not execute at all because of limited computer memory. Thus, while this method has a great potential, we currently need to develop alternate means of fast synthesis for large problem instances. The above-mentioned

heuristics were developed for that purpose to exploit the structure of the synthesis problem.

Using results obtained during makespan optimizations through ILP-based synthesis, we determined that one can optimize architectures by initially allocating best processor and network resources for each node and edge, followed by making decisions on processor sharing to minimize inter-task communication costs. This being a greedy-like approach, we formally proved using matroid theory that there exist a linear time greedy algorithm for the automated synthesis problem when there are no resource constraints, the latter being defined as cases where larger resource capacities do not lead to better solutions. Otherwise, dependencies between synthesis subproblems necessitate search heuristics. Nevertheless, we showed that greedy algorithms likely require a full schedulability analysis, therefore given memory implications, we developed three different heuristics that attempt to optimize using a partial list. These heuristics differ on whether and how local search is conducted.

The first heuristic uses a one-level backtracking strategy when making processor sharing decisions when more expensive solutions are encountered. The second uses a restarting scheme to do the same by un-mapping and re-mapping tasks associated with expensive data traffic. The third heuristic is a one pass greedy-like allocation which is very similar in nature to a pure greedy algorithm. We evaluated the heuristics using the same set of applications against ILP-based synthesis both in terms of runtime and quality of results. All algorithms are up to three orders of magnitude faster versus ILP-based synthesis, requiring only a few seconds even for the largest problem instances studied. However, none of them is overall superior in terms of quality of synthesized architectures. In most synthesis scenarios conducted, at least one of the heuristic delivered results close to or equal to those obtained using ILP-based synthesis. Therefore, in order to obtain both fast synthesis and acceptable quality of results, we proposed a voting scheme, where all three heuristics are used, and the best result is selected.

All synthesis methods developed are integrated in a command line tool whose interface is described in Appendix B. The integration of this tool in the design platform PinHat enables us to meet research objectives formulated in Chapter 1 in order to complete the design flow of flexible CMP systems from parallel programs to FPGA bitstream.

6.2 Outlook

While the tool developed in the course of this research is already useful in its current form, the work conducted led to a number of open problems which need to be addressed in future work. The first is accounting for temporal information with respect to data traffic during synthesis in order to reduce resource requirements or to achieve better throughput or application runtime. Currently, worst case scenarios are assumed. Second, it is important to incorporate energy and routing models as described in Chapter 1 so as to explore energy-performance trade-offs. With respect to ASP-based synthesis, it is interesting to investigate how schedulability analysis can be integrated in the solver or ASP programs, because that would alleviate memory limitations. Alternately, improvements in developed heuristics targeting better local search techniques would be equally effective.

Finally, integration of cycle accurate processor and NoC simulators as well as NoC analysis tools in the overall design flow is a vital component towards a more mature tool suite, alongside the development of a rich IP library. We believe that, with a continued proliferation of better FPGA devices, the tool and methods developed herein will be useful for coping with increased design complexity in pursuit of better architectures to meet ever increasing performance requirements.

A Glossary and Notation

A.1 Glossary

| | |
|-------|--|
| ASIC | Application-Specific Integrated Circuit |
| ASIP | Application-Specific Instruction-set Processor |
| ASP | Answer Set Programming |
| CMP | Chip Multi-Processor (systems) |
| CP | Critical Path |
| DAG | Directed Acyclic Graph |
| DLL | Delay Locked Loop |
| DP | Domain-specific Processor |
| DPLL | Davis-Putnam-Logemann-Loveland (algorithm) |
| DSE | Design Space Exploration |
| FFT | Fast Fourier Transform |
| FIR | Finite Impulse Response (filter) |
| FPGA | Field Programmable Gate Array |
| FSL | Fast Simplex Link |
| GPP | General Purpose Processor |
| ILP | Integer Linear Programming |
| IP | Intellectual Property (core) |
| MB | Microblaze (processor) |
| MPI | Message Passing Interface |
| MPMD | Multi-Program Multi-Data |
| MIPS | Million of Instructions per Second |
| MPSD | Multiple Programs Single (Data Stream) |
| MPSoC | Multi-Processors System-On-Chip |
| NoC | Network On-Chip |
| OPB | On-chip Peripheral Bus |
| PN | Pseudo-random Number |
| PPC | Power PC (processor) |
| QoS | Quality of Service |
| rASIP | reconfigurable ASIP |
| RISC | Reduced Instruction-Set Computer |
| RMA | Rate Monotonic Analysis |
| RPS | Resolution Proof System (for SAT problems) |
| RTL | Register Transfer Level |

| | |
|-------|---|
| SAT | Boolean Satisfiability (problem) |
| SDR | Software Defined Radio |
| SIMD | Single Instruction Multiple Data |
| SPMD | Single Program Multiple Data (Streams) |
| VLIW | Very Large Instruction Word Processor |
| WCDMA | Wide-band Code Division Multiple Access |
| WLAN | Wireless Local Area Network |

A.2 Notation

| | |
|----------------------------|---|
| $\alpha_{i_1 i_2 j_1 j_2}$ | An auxiliary decision variable for a pair of communicating tasks mapped on two processors |
| a_j | The area of a processor J_j |
| A_{PE} | The maximum area on FPGA for mapping processing elements |
| A_{net} | The maximum area on FPGA for mapping networks |
| A_k | Area of C_k |
| A | Total FPGA area |
| B_{i_1, i_2} | The number of data blocks transferred between two communicating tasks |
| χ | An element of an independent subset of a matroid |
| c_i | A SAT clause |
| \mathfrak{C} | A set of networks in IP library |
| C_k | A network with an index k |
| $D_{i_1 i_2}$ | The amount of data transferred between two communicating tasks |
| D_i | Deadline of a task |
| E | Number of edges in application graph |
| F_{lj} | Coefficient for schedulability of G_l on J_j |
| G_l | A group of task with an index l |
| g_l | Coefficient indicating if a group contains a task on critical path |
| γ_{lj} | A decision variable for mapping G_l on J_j where the G_l is the largest mapped group on J_j |
| \mathfrak{J} | A set of tasks |
| I_i | A task with an index i |
| J_j | A processor with an index j |
| $\tilde{\mathfrak{J}}$ | A set of processors in IP library |
| K' | $K + 1$ |
| K | \mathfrak{C} |
| $\lambda_{i_1 i_2}$ | An auxiliary decision variable for a pair of communicating tasks mapped on different processors |
| L_k | The transfer latency for network C_k |
| L | An independent subset of a matroid |

| | |
|--------------------|---|
| M_k | Capacity of C_k |
| M | $ \mathfrak{J} $ |
| m | $M - 1$ |
| \mathcal{M}_{lj} | Decision variable for mapping G_l on J_j |
| \mathfrak{M} | A matroid |
| n | $N - 1$ |
| N | $ \mathfrak{J} $ |
| O_j | Switching overhead due to clock handler on J_j |
| p_k | The probability of arbitration in C_k |
| p_i | A literal in SAT clause |
| $\mathcal{P}(I)$ | Power set of I |
| r | The response of a task in preemptive scheduling |
| S | A finite set |
| s_j | The size of the program memory of processor J_j |
| s_{ij} | The size of task I_i in the program memory of processor J_j |
| τ_k | The bound on arbitration time in C_k |
| T_{switch} | Task switching overhead |
| T_{sl_j} | The number of task switching for G_l on J_j |
| $T_{s_{ij}}$ | The number of task switching for a task on J_j |
| t_j | The cost of task switching on processor J_j |
| T'_{net} | Total cost in time due to networks under makespan extension |
| T'_{sl_j} | Reduced number of context switchings for G_l on J_j under makespan optimization |
| T'_{switch} | Scheduling overhead in makespan mode |
| T_{ij} | Duration of task I_i on J_j |
| T_{clock_j} | Interval of a clock handler for OS/kernel on J_j |
| \mathcal{T}_i | Period of a task |
| φ | A Boolean formula |
| v_j | Decision variable for a virtual processor for J_j |
| X_i | Coefficient indicating whether a task is on critical path |
| x_{ij} | A decision variable for a task I_i mapped on processor J_j |
| y_k | Decision variable for a network C_k |
| $z_{ki_1i_2}$ | A decision variable for C_k allocated for a pair of communicating tasks |
| $Z_{i_1i_2}$ | Coefficient indicating whether an edge is on critical path |

B The Synthesis Tool

This is a command line tool implemented in C++. There are four modes: mode 1 is ILP-based synthesis which generates model data and runs the solver `lp_solve` by way of the library `lpsolve55`, mode 2 accepts previously generated model data and runs the same solver, mode 3 is ASP-based synthesis which generates ASP programs and externally invoke the solver `clasp`, and mode 4 uses one of the three synthesis heuristics. The usage for mode 1, 3 and 4 is:

```
ilpf [-h] [-d data] [-l log] [-t timeout] -c clog -m model
-p processors -n networks -S sizes -C cycles [-P powerSetFile]
-I taskInfoFile [-M] [-A | -G | -G+r | -G+g | -G+p]
```

where [] are optional arguments.

-h: Prints help contents.

-d: Specifies the name of the GNU Mathprog ILP data file for output. This file is generated after the formulation, and can be passed as argument in mode 2. If the file is not specified, a default file `data.ilp` is generated instead. The name of the data file (regardless if passed or default) is post fixed with pass number if in makespan mode (e.g. `data.ilp_pass.3`).

Note: GNU Mathprog format allows an ILP model to be split into a generic model file, and a problem instance specific data file. Both files are read by the solver using an API defined in `xliMathProg` library. Therefore, the synthesis tool requires both dynamically linked libraries `lpsolve55` and `xliMathProg`.

-l: Specifies the name of the log file for the ILP solver. If the file is not specified, solver output will be printed to `stdout` only. A separate log file `synthesis.rpt` is generated for all other printouts from the formulator.

-t: Specifies the solver time out in seconds. The solver will abort after this time. If the parameter is not specified, a default value of 18000 seconds is used. Specifying this parameter influences the tool in this way:

- If an optimum solution is found, the solution is written out to a file `data.lp_solution`, where `data` is the output data file.
- If a user specifies the parameter and a feasible solution is found, then the solution is written out to a file `data.lp_solution.0`.
- If a default timeout is used, and a feasible solution is found, then the tool attempts to find an optimum solution by changing solver settings and restarting the solver. A maximum of three further attempts are done. Suboptimum solutions are written out to a file `data.lp_solution_n`, where `n` is the number of an attempt (0,1,2 or 3). Solver settings in further attempts are:

Attempt 1:

Table B.1: Supported Kernel IDs

| ID | Kernel | Remarks |
|----|----------------|--|
| 0 | Cyclic | This is a basic cyclic executive. There is no scheduling, so all group of tasks can be mapped on the processor, and there is no switching cost. |
| 1 | Timed | This model assumes that each task will be activated by interrupts. i.e. if an interrupt occurs, one or more tasks in a task group will execute. The group has a feasible schedule if the sum of all task durations $\sum T_{ij}$ is less than the interrupt interval. The switching cost is zero because there is no preemption. |
| 2 | Fixed-Priority | Calculates switching cost and schedule feasibility according to Algorithm 1. |
| 3 | Round Robin | This is a round robin preemptive model. Any group has a feasible schedule. Switching cost is computed as $\lceil \sum T_{ij} / \text{clock interrupt interval} \rceil$. |

The transfer latency for a word for the network is as described in Section 3.2.1.4 (parameter L_k). See the paragraph below on units.

The network area on FPGA is used for area constraint. Any appropriate metric and units can be given as these are not further interpreted. However, units and metrics for all networks must be consistent, and they also must be consistent to those given under processor parameters.

The capacity specifies the maximum number of processors that can be attached to the network as discussed in Section 3.2.1.4 (parameter M_k).

The name of instances are not interpreted in the formulator: their use is to help the designer differentiate between different instances. There is no restriction other than that a name must be representable as a C++ string.

Multi-hop networks are specified using the same format. Parameters given should reflect QoS guarantees as discussed in Section 3.2.1.4.

All time units must be the same as those that will be derived from the clock frequency in the processor file. The tool performs no checking. If for instance the time unit for the parameters given here are in seconds, but the frequency is in MHz, then results will be falsely biased against network usage.

-S: Specifies the name of the file containing task sizes on processors. Sizes given in this file must be consistent with the size parameter in the processor file (parameters s_{ij}). The tool performs no checks. The format is as shown in Listing 3. Note that the format is essentially a table, where each row specifies the size of a given task on a given processor. Following the convention in ILP formulation in Chapter 3, processor indices are J0, J1, etc, whereas task indices are I0, I1, etc. Also note that this file specifies two additional parameters: FPGA size constraints for processing elements A_{PE} and for networks A_{net} . These two constraints must be consistent with area parameters specified for processors and networks in corresponding files. The tool performs no checks. The file is automatically generated by PinHat.

-C: Specifies the name of the file containing task durations on processors (parameters T_{ij}) as shown in Listing 4.

Note that the format is essentially a table, where each row specifies the duration of a task on a processor. While not enforced, time units must be consistent with those given or implied in

```

/* specify the area constraints */
param A_pe := 31584;
param A_net := 31584;
/* specify the table with task size on
each processor */
param taskSize : J0 J1 J2 J3 J4 J5 :=
    I0 5 5 5 5 5 5
    I1 5 5 5 5 5 5
    I2 5 5 5 5 5 5 ;

```

Listing 3: Format for task sizes on processors

```

/* specify the table with task durations on
each processor */
param procTime : J0 J1 J2 J3 J4 J5 :=
    I0 6032e-9 6032e-9 1034e-9 1034e-9 11e-9 11e-9
    I1 3504e-9 3504e-9 601e-9 601e-9 601e-9 601e-9
    I2 909e-9 909e-9 909e-9 156e-9 156e-9 156e-9 ;

```

Listing 4: Format for task cycles on processors

network and processor file. The information in this file is also used for computing task priorities and in schedulability analysis. The file is automatically generated by PinHat.

- P:** Specifies the name of the file containing the power set $\mathcal{P}(I)$ of the task set $I = \{I_0, \dots, I_n\}$ (see Section 3.2.2.4). Listing 5 shows the format (only the last 5 lines for $N = 9$ are shown). This file is automatically generated by PinHat, and is only required for ILP and ASP synthesis modes.

```

I0 I1 I2 I4 I5 I6 I7 I8 I9
I0 I1 I3 I4 I5 I6 I7 I8 I9
I0 I2 I3 I4 I5 I6 I7 I8 I9
I1 I2 I3 I4 I5 I6 I7 I8 I9
I0 I1 I2 I3 I4 I5 I6 I7 I8 I9

```

Listing 5: Format for power set

- I:** Specifies the name of the file containing task information (period, rate and deadline) as outlined in Listing 6. The file is generated by PinHat based on user input.

The number of times a task has ran allows the user to influence if the parameter T_{ij} should be used for one period, or for the total duration of the application. The formulator always treats T_{ij} as the time for one period. To obtain the total duration, T_{ij} is multiplied by the number of times the task has ran. Setting the number of runs to 1 means that the optimization result is based on one period for that task. Specifying a number less than 1 will cause the formulator to abort with an error.

The deadline is used in schedulability analysis and in assigning priorities. This parameter is

```

I0 65 1 65 10240
I1 4160 1 4160 159
//|  |  |  |  |__number of times the task ran
//|  |  |  |_____deadline
//|  |  |_____rate
//|  |_____period
//|_____task ID

```

Listing 6: Format for task information

ignored if the kernel model does not consider deadlines. The time unit must be consistent with those given in network and processor file.

The rate is a reserved parameter currently not in use.

The period is also used in schedulability analysis and in assigning priorities (see Section 3.2.2). The time units must be consistent with those given in network and processor file.

-M: Toggles the Makespan mode. The application graph must be a DAG, and it must be connected. Otherwise, the makespan is undefined, and the formulator aborts with an error.

-A : Toggles synthesis in ASP mode.

The sizes should be scaled such that integral numbers can be specified when synthesizing in ASP mode because the tool will round up the figures entered in order to meet solver’s integrality restrictions (see Section 4.3).

In this mode, the synthesis tool generates ASP programs and invokes Clingo for grounding and resolution. No options are used. ASP programs are written out to `<data file base name>.lp_pass_n`, where the base name of the data file is optionally specified by `[-d]`, and `n` is the pass number when synthesizing in makespan mode. Clingo enumerates found stable models, which are piped to `<data file name>_pass_n.asp_solution`. The last model in the file is the best one. Additionally, a report file `synthesis.rpt` is generated.

-G Toggles greedy heuristic mode. All options apply as in `-A` mode, but in this case, the powerset file can be omitted since scheduling analysis is computed as needed, rather than completely in advance. This mode produces the report file `synthesis.rpt` only, which in this case also contains the solution.

Greedy mode can optionally be supplemented by “plus” options which further specify one of the three heuristics.

-G+r This is the default mode, equivalent to `-G`. The default mode uses the “replace and search” heuristic (see Section 5.2).

-G+g This uses the “group growing” heuristic (see Section 5.3).

-G+p This uses the “priority” heuristic (see Section 5.4).

The usage for mode 2 is:

```
ilpf -s -d data -m model
```

where switches `-d` and `-m` have the same meaning as discussed above. The switch `-s` toggles mode 2.

Bibliography

- [1] M. Gries, “Methods for evaluating and covering the design space during early design development,” *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, 2004. [1](#), [19](#), [21](#)
- [2] T. T. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*. Cambridge, MA, USA: MIT Press, 1990. [iv](#), [94](#), [95](#), [96](#), [97](#)
- [3] G. E. Moore, “Cramming more components onto integrated circuits,” pp. 56–59, 2000. [1](#)
- [4] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, “Leakage current: Moore’s law meets static power,” *Computer*, vol. 36, pp. 68–75, Dec. 2003. [1](#)
- [5] D. P. John Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3 ed., 2005. [1](#)
- [6] H. Ishebabi, G. Ascheid, H. Meyr, O. Atak, A. Atalar, and E. Erikan, “An efficient parallelization technique for high throughput FFT-ASIPs,” (Kos, Greece), May 2006. [1](#)
- [7] C. Bobda, *Introduction to Reconfigurable Computing*. Springer, 2007. [2](#), [3](#), [4](#)
- [8] G. Estrin and R. Turn, “Automatic assignment of computations in a variable structure computer system,” *Electronic Computers, IEEE Transactions on*, vol. EC-12, pp. 755–773, Dec. 1963. [2](#)
- [9] C. Bobda, T. Haller, F. Muehlbauer, D. Rech, and S. Jung, “Design of adaptive multiprocessor on chip systems,” in *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, (New York, NY, USA), pp. 177–183, ACM, 2007. [2](#), [4](#)
- [10] W.-T. Zhang, L.-F. Geng, D.-L. Zhang, G.-M. Du, M.-L. Gao, W. Zhang, and Y.-H. T. Ning Hou, “Design of heterogeneous MPSoC on FPGA,” in *ASICON '07: 7th International Conference on*, pp. 102–105, 2007. [2](#)
- [11] M. Saldana, D. Nunes, E. Ramalho, and P. Chow, “Configuration and programming of heterogeneous multiprocessors on a Multi-FPGA system using TMD-MPI,” *IEEE International Conference on Reconfigurable Computing and FPGA's*, pp. 1–10, 2006. [2](#), [24](#)
- [12] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, “Atlas: a chip-multiprocessor with transactional memory support,” in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, (San Jose, CA, USA), pp. 3–8, EDA Consortium, 2007. [2](#)
- [13] M. Nikitovic and M. Brorsson, “An adaptive chip-multiprocessor architecture for future mobile terminals,” in *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, (New York, NY, USA), pp. 43–49, ACM, 2002. [3](#)

- [14] Y. Tsividis, *Operation and modeling of the MOS transistor*. New York, NY, USA: McGraw-Hill, Inc., 1987. 4
- [15] C. Hu, *Device and Technology Impact on Low Power Electronics*. Kluwer Academic, 1996. 4
- [16] D. B. Noonburg and J. P. Shen, "A framework for statistical modeling of superscalar processor performance," *hpca*, p. 298, 1997. 9
- [17] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, "Automated energy/performance macro-modeling of embedded software," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, (New York, NY, USA), pp. 99–102, ACM, 2004. 9
- [18] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. 14, 17
- [19] W. J. Dally, "Virtual-channel flow control," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 60–68, 1990. 15, 17
- [20] A. Maheshwari, W. Burleson, and R. Tessier, "Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, pp. 299–311, March 2004. 16
- [21] C. Grecu, A. Ivanov, R. Pande, A. Jantsch, E. Salminen, U. Ogras, and R. Marculescu, "Towards open network-on-chip benchmarks," *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pp. 205–205, May 2007. 17
- [22] M. Ehrgott, *Multicriteria Optimization*. Germany: Springer, 2 ed., 2005. 19
- [23] J. Andersson, "A survey of multiobjective optimization in engineering design," 2000. 19
- [24] V. Pareto, "Cours d'economie politique," 1896. 19, 20
- [25] K. Klamroth, J. Tind, and S. Züst, "Integer programming duality in multiple objective programming," *J. of Global Optimization*, vol. 29, no. 1, pp. 1–18, 2004. 20
- [26] K. Sarrigeorgidis and J. Rabaey, "A scalable configurable architecture for advanced wireless communication algorithms," *J. VLSI Signal Process. Syst.*, vol. 45, no. 3, pp. 127–151, 2006. 21
- [27] V. Lapinskii, *Algorithms for compiler-assisted design space exploration of clustered VLIW ASIP datapaths*. PhD thesis, 2001. Supervisor-Margarida F. Jacome. 21
- [28] M. Gries and K. Keutzer, *Building ASIPs: The Mescal Methodology*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. 21
- [29] A. Chattopadhyay, X. Chen, H. Ishebabi, R. Leupers, G. Ascheid, and H. Meyr, "High-level modelling and exploration of coarse-grained re-configurable architectures," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, (New York, NY, USA), pp. 1334–1339, ACM, 2008. 21
- [30] D. Fischer, J. Teich, M. Thies, and R. Weper, "Efficient architecture/compiler co-exploration for ASIPs," in *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, (New York, NY, USA), pp. 27–34, ACM, 2002. 21

- [31] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl, "A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, (Washington, DC, USA), p. 21256, IEEE Computer Society, 2004. [21](#), [22](#)
- [32] F. Fummi, S. Martini, G. Perbellini, and M. Poncino, "Native ISS-SystemC integration for the co-simulation of multi-processor SoC," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, (Washington, DC, USA), p. 10564, IEEE Computer Society, 2004. [21](#)
- [33] S. Chakraborty, S. Kunzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," *date*, vol. 01, p. 10190, 2003. [22](#)
- [34] K. Goossens, J. Dielissen, and A. Radulescu, "Aethereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test*, vol. 22, no. 5, pp. 414–421, 2005. [22](#)
- [35] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, (Washington, DC, USA), p. 20890, IEEE Computer Society, 2004. [22](#)
- [36] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28tflops network-on-chip in 65nm cmos," *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 98–589, Feb. 2007. [22](#)
- [37] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *J. Syst. Archit.*, vol. 50, no. 2-3, pp. 105–128, 2004. [22](#)
- [38] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to constrained mapping and routing on network-on-chip architectures," in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 75–80, ACM, 2005. [22](#)
- [39] X. Zhu, W. Qin, and S. Malik, "Modeling operation and microarchitecture concurrency for communication architectures with application to retargetable simulation," in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 66–71, ACM, 2004. [22](#)
- [40] C. Grecu and M. Jones, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 1025–1040, 2005. Student Member-Partha Pratim Pande and Senior Member-Andre Ivanov and Senior Member-Resve Saleh. [22](#)
- [41] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant network-on-chip architectures," in *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 93–104, IEEE Computer Society, 2006. [22](#)
- [42] M. Moadeli, A. Shahrabi, and W. Vanderbauwhede, "Analytical modelling of communication in the rectangular mesh noc," *Parallel and Distributed Systems, 2007 International Conference on*, vol. 2, pp. 1–8, Dec. 2007. [22](#)
- [43] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comput. Surv.*, vol. 38, no. 1, p. 1, 2006. [22](#)

- [44] J. Xi and P. Zhong, "A transaction-level NoC simulation platform with architecture-level dynamic and leakage energy models," in *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, (New York, NY, USA), pp. 341–344, ACM, 2006. [22](#)
- [45] P. Wolkotte, P. Holzspies, and G. Smit, "Fast, accurate and detailed NoC simulations," *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pp. 323–332, May 2007. [22](#)
- [46] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore SoCs," *dac*, p. 789, 2002. [22](#)
- [47] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "UML-based multiprocessor SoC design framework," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 281–320, 2006. [22](#)
- [48] D. Quinn, B. Lavigueur, G. Bois, and M. Aboulhamid, "A system level exploration platform and methodology for network applications based on configurable processors," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, (Washington, DC, USA), p. 10364, IEEE Computer Society, 2004. [22](#)
- [49] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Extending the transaction level modeling approach for fast communication architecture exploration," *DAC*, pp. 113–118, 2004. [22](#)
- [50] D. Bertozzi and A. Jalabert, "NoC synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, 2005. Student Member-Srinivasan Murali and Student Member-Rutuparna Tamhankar and Student Member-Stergios Stergiou and Member-Luca Benini and Fellow-Giovanni De Micheli. [22](#)
- [51] R. B. Mouhoub and O. Hammami, "MOCDEX: Multiprocessor on chip multiobjective design space exploration with direct execution," *EURASIP Journal on Embedded Systems*, vol. 2006, 2006. [22](#)
- [52] S. Kang and R. Kumar, "Magellan: A search and machine learning-based framework for fast multi-core design space exploration and optimization," *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 1432–1437, March 2008. [23](#)
- [53] B. K. Dwivedi, A. Kumar, and M. Balakrishnan, "Automatic synthesis of system on chip multiprocessor architectures for process networks," in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 60–65, ACM, 2004. [23](#)
- [54] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 273–278, ACM, 2005. [23](#)
- [55] G. Beltrame, D. Bruschi, D. Sciuto, and C. Silvano, "Decision-theoretic exploration of multiprocessor platforms," in *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 205–210, ACM, 2006. [23](#)
- [56] B. Meyer and D. Thomas, "Rethinking automated synthesis of MPSoC architectures," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–6, March 2007. [23](#)

- [57] R. Dick, "Multiobjective synthesis of low-power real-time distributed embedded systems," 2002. [23](#)
- [58] C. Zhu, Z. P. Gu, R. P. Dick, and L. Shang, "Reliable multiprocessor system-on-chip synthesis," in *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, (New York, NY, USA), pp. 239–244, ACM, 2007. [23](#)
- [59] C. Lee and S. Ha, "Hardware-software cosynthesis of multitask MPSoCs with real-time constraints," in *6th International Conference on ASIC*, vol. 2, pp. 919–924, 2005. [23](#)
- [60] S. Ha, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "Hardware-software codesign of multimedia embedded systems: the peace," *rtcsa*, pp. 207–214, 2006. [23](#)
- [61] D. Arora, A. Raghunathan, S. Ravi, M. Sankaradass, N. K. Jha, and S. T. Chakradhar, "Software architecture exploration for high-performance security processing on a multiprocessor mobile SoC," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*, (New York, NY, USA), pp. 496–501, ACM Press, 2006. [24](#)
- [62] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: A high-end reconfigurable computing system," *IEEE Des. Test*, vol. 22, no. 2, pp. 114–125, 2005. [24](#)
- [63] K. Huang, S. il Han, K. Popovici, L. Brisolaro, X. Guerin, L. Li, X. Yan, S. Ik Chae, L. Carro, and A. A. Jerraya, "Simulink-based MPSoC design flow: case study of motion-JPEG and H.264," in *DAC '07: Proceedings of the 44th annual conference on Design automation*, (New York, NY, USA), pp. 39–42, ACM, 2007. [24](#)
- [64] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, 2006. [24](#)
- [65] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 7, pp. 711–726, 2004. [24](#)
- [66] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1523–1543, 2000. [24](#)
- [67] P. Pop, *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. PhD thesis, Linköping University, 2003. [24](#)
- [68] D. L. Rhodes and W. Wolf, "Co-synthesis of heterogeneous multiprocessor systems using arbitrated communication," in *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, (Piscataway, NJ, USA), pp. 339–342, IEEE Press, 1999. [24](#)
- [69] C. Erbas, S. Cerav-Erbas, and A. Pimentel, "Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design," *Evolutionary Computation, IEEE Transactions on*, vol. 10, pp. 358–374, June 2006. [24](#)
- [70] M. Bao, A. Andrei, P. Eles, and Z. Peng, "Temperature-aware task mapping for energy optimization with dynamic voltage scaling," *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, pp. 1–6, April 2008. [24](#)

- [71] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano, "Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, (3001 Leuven, Belgium, Belgium), pp. 3–8, European Design and Automation Association, 2006. 24
- [72] S. Carta, F. Mereu, A. Acquaviva, and G. Micheli, "Migra: A task migration algorithm for reducing temperature gradient in multiprocessor systems on chip," *System-on-Chip, 2007 International Symposium on*, pp. 1–6, Nov. 2007. 24
- [73] P. K. Holzenspies, J. L. Hurink, J. Kuper, and G. J. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSOC)," *Design, Automation and Test in Europe, 2008. DATE '08*, pp. 212–217, March 2008. 24
- [74] Y. Guo, D. McCain, J. Cavallaro, and A. Takach, "Rapid industrial prototyping and SoC design of 3G/4G wireless systems using an HLS methodology," *EURASIP Journal on Embedded Systems*, vol. 2006, 2006. 24
- [75] H. G. Lee, U. Y. Ogras, R. Marculescu, and N. Chang, "Design space exploration and prototyping for on-chip multimedia applications," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*, (New York, NY, USA), pp. 137–142, ACM Press, 2006. 24
- [76] Y. Aoudni, K. Loukil, G. Gogniat, J. L. Philippe, and M. Abid, "Mapping SoC architecture solutions for an application based on PACM model," in *IEEE International Symposium on Industrial Electronics*, 2006. 24
- [77] C. Guret, C. Prins, M. Sevaux, and S. Heipcke, *Applications of Optimization with XpressMP*. Dash Optimization Ltd, 2002. 32, 34
- [78] <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>. 38
- [79] <http://www.mcs.anl.gov/research/projects/mpich2/>. 39
- [80] <http://lpsolve.sourceforge.net/5.5/MathProg.htm>. 41
- [81] <http://lpsolve.sourceforge.net/5.5/>. 41
- [82] T. Schmidl and T. Cox, "Robust frequency and timing synchronization for OFDM," *IEEE Transactions on Communications*, vol. 45, no. 12, pp. 1613–1621, 1997. 55
- [83] J. Heiskala and J. Terry, *OFDM Wireless LANs: A Theoretical and Practical Guide*. SAMS publishing, 2002. 55
- [84] R. V. Nee and R. Prasad, *OFDM for Wireless Multimedia Communications*. Artech House, 2000. 55
- [85] C.-F. Hsu, Y.-H. Huang, and T.-D. Chiueh, "Design of an OFDM receiver for high-speed wireless LAN," in *IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 558–561, 2001. 55
- [86] F. Adachi, M. Sawahashi, and H. Suda, "Wideband DS-SS-CDMA for next-generation mobile communications systems," *IEEE Communications Magazine*, vol. 36, pp. 56–69, 1998. 55
- [87] H. Schulze and C. Lüders, *Theory and Application of OFDM and CDMA*. Wiley, 2005. 55
- [88] K. Kettunen, "Enhanced maximal ratio combining scheme for RAKE receivers in WCDMA mobile terminals," in *Electronics Letters*, vol. 37, pp. 522–524, April 2001. 55

- [89] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973. [57](#), [59](#)
- [90] A. Burns, "Preemptive priority based scheduling: An appropriate engineering approach," Tech. Rep. 214, 1994. [57](#)
- [91] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, pp. 475–480, 1995. [59](#)
- [92] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors:," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996. [63](#)
- [93] Y. Tian, Y. Gu, E. Ekici, and F. Ozguner, "Dynamic critical-path task mapping and scheduling for collaborative in-network processing in multi-hop wireless sensor networks," *icppw*, pp. 215–222, 2006. [63](#)
- [94] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, pp. 269–271, 1959. [64](#)
- [95] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "clasp: A conflict-driven answer set solver," in *Proceedings of the Ninth International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'07)*, vol. 4483, pp. 260–265, springer, 2007. [73](#)
- [96] A. Morgado, P. Matos, V. Manquinho, and J. Marques-Silva, "Counting models in integer domains," *Lecture Notes in Computer Science*, vol. 4121, pp. 410–423, 2006. [73](#)
- [97] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, pp. 156–173, April 2005. [73](#)
- [98] N. Een and N. Sorensson, "Translating pseudo-boolean constraints into sat," in *Journal on Satisfiability, Boolean Modeling and Computation*, 2006. [73](#)
- [99] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM*, vol. 12, no. 1, pp. 23–41, 1965. [73](#), [74](#)
- [100] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960. [74](#)
- [101] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, (Piscataway, NJ, USA), pp. 279–285, IEEE Press, 2001. [76](#)
- [102] H. Zhang, "SATO: An efficient propositional prover," in *CADE-14: Proceedings of the 14th International Conference on Automated Deduction*, (London, UK), pp. 272–275, Springer-Verlag, 1997. [76](#)
- [103] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *DAC '01: Proceedings of the 38th conference on Design automation*, (New York, NY, USA), pp. 530–535, ACM, 2001. [77](#)
- [104] P. Simons, I. Niemelá, and T. Sojininen, "Extending and implementing the stable model semantics," *Artif. Intell.*, vol. 138, no. 1-2, pp. 181–234, 2002. [77](#)

- [105] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, “A user’s guide to gringo, clasp, clingo, and iclingo,” November 2008. [77](#), [78](#), [80](#), [81](#)
- [106] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, “T.: Conflict-driven answer set solving,” in *In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07*, pp. 386–392, MIT Press, 2007. [77](#)
- [107] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994. [94](#)
- [108] D. H. Wolpert and W. G. Macready, “No free lunch theorems for optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 67–82, 1997. [95](#)