Méthodologie de prototypage rapide pour systèmes embarqués parallèles : modélisation des systèmes et amélioration des heuristiques d'ordonnancement de tâches

Pengcheng Mu

▶ **To cite this version:**

Pengcheng Mu. Méthodologie de prototypage rapide pour systèmes embarqués parallèles : modélisation des systèmes et amélioration des heuristiques d'ordonnancement de tâches. Modélisation et simulation. INSA de Rennes, 2009. Français. NNT : . tel-00429417

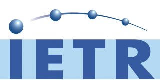
HAL Id: tel-00429417
https://theses.hal.science/tel-00429417

Submitted on 2 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre :



Ecole Doctorale MATISSE

THESE

présentée devant

L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES

pour obtenir le grade de

DOCTEUR

spécialité : *Traitement du Signal et de l'Image*

Rapid Prototyping Methodology for Parallel Embedded Systems

Advanced System Model and Improved Task Scheduling Heuristics

par

MU Pengcheng

Soutenance prévue le 07/07/2009 devant la commission d'Examen

Composition du jury:

Rapporteurs

BOURENNANE El-Bay	Professeur des Universités à l'Université de Bourgogne
HOUZET Dominique	Professeur des Universités à l'INPG de Grenoble

Examinateurs

GRANDPIERRE Thierry	Professeur Associé à l'ESIEE, Noisy-le-Grand
VERDIER François	Maître de Conférences (HDR) à l'ETIS
COUSIN Jean-Gabriel	Maître de Conférences à l'INSA de Rennes
NEZAN Jean-François	Maître de Conférences à l'INSA de Rennes

Directeur de thèse

RONSIN Joseph	Professeur des Universités à l'INSA de Rennes

Institut d'Electronique et de Télécommunications de Rennes
Institut National des Sciences Appliquées de Rennes
Université Européenne de Bretagne

Contents

Contents iii

Introduction 1

1 Rapid Prototyping and Hardware/Software Co-design 5

1.1 Introduction 5

1.2 Design FPGA based MPSoC with AAA Rapid Prototyping Methodology 6

1.2.1 AAA Rapid Prototyping Methodology and SynDEx 6

1.2.2 Rapid Prototyping for Multi-MicroBlaze Systems on FPGA . 8

1.2.3 SynDEx-Ic Tool 12

1.3 Tools for HDL Code Generation 13

1.3.1 GAUT: A High-Level Synthesis Tool 14

1.3.2 Open Dataflow Framework 16

1.3.3 Comparison between GAUT and OpenDF 21

1.4 An Eclipse-Based Open Source Rapid Prototyping Framework 22

1.4.1 Graphiti: A Generic Graph Editor for Editing Architectures, Algorithms and Workflows 23

1.4.2 SDF4J: A Java Library for Algorithm Dataflow Graph Transformation 25

1.4.3 PREESM: A Complete Framework for Hardware/Software Co-design . . . 27
1.5 Conclusion . . . 28

2 Graph Models for Parallel Embedded Systems 29
2.1 Introduction . . . 29
2.2 Algorithm Model . . . 30
2.2.1 Dataflow Model . . . 30
2.2.2 DAG Model . . . 34
2.2.3 DAG Properties . . . 36
2.3 Architecture Model . . . 43
2.3.1 Parallel Architectures . . . 43
2.3.2 Advanced Architecture Model . . . 45
2.3.3 Architecture Specification with IP-XACT Standard . . . 51
2.4 Conclusion . . . 53

3 Task Scheduling in Parallel Embedded Systems 55
3.1 Introduction . . . 55
3.2 General Task Scheduling . . . 56
3.2.1 Without/With Communication Costs . . . 56
3.2.2 Scheduling Methodologies . . . 58
3.2.3 Advanced Techniques . . . 60
3.3 Task Scheduling with Advanced Architecture Model . . . 63
3.3.1 Routing with Architecture Model . . . 63
3.3.2 Scheduling with Advanced Architecture Model . . . 65
3.3.3 Causality Conditions . . . 68
3.3.4 Scheduling Conditions . . . 69
3.4 Task Scheduling with Topology Graph Model . . . 71
3.4.1 Topology Graph Model . . . 71
3.4.2 Scheduling with Communication Contention . . . 73
3.5 Conclusion . . . 75

4 List Scheduling with Communication Contention 77
4.1 Introduction . . . 77
4.2 Node Levels with Communication Contention . . . 78
4.3 List Scheduling Heuristics . . . 82

4.3.1 Static List Scheduling Heuristic 82
4.3.2 Dynamic List Scheduling Heuristic 85
4.4 Experimental Results 86
4.4.1 Comparison with an Example 87
4.4.2 Comparison with Randomly Generated DAGs 89
4.5 Analysis of Time Complexity 95
4.6 Conclusion 97

5 Advanced List Scheduling Methods 99
5.1 Introduction 99
5.2 Processor Selection with Critical Child 100
5.3 Node and Edge Scheduling with Communication Delay 102
5.3.1 Node Scheduling 102
5.3.2 Edge Scheduling 103
5.4 Advanced List Scheduling Heuristics 104
5.5 Experimental Results 105
5.5.1 Comparison with an Example 106
5.5.2 Comparison with Randomly Generated DAGs 108
5.6 Time Complexity of Advanced List Scheduling Heuristics 114
5.7 Conclusion 118

Conclusions and Prospects 119

A IP-XACT Code of Advanced Architecture Model 123
A.1 TI's C6474 DSP 123
A.2 Xilinx's FPGA-based MPSoC 128

List of Figures 135

List of Tables 139

List of Algorithms 141

Personal Publications 143

Bibliography 145

Index 157

Abstract **162**

Introduction

Context

Embedded systems are pervasive around our everyday life. An embedded system is an electronic system dedicated to specific applications. These systems especially exist as consumer electronics: PDAs, mp4 players, mobile phones, videogame consoles, digital cameras, DVD players, GPS receivers, etc. Most of these systems consist in digital signal processing and/or image processing applications that require a lot of processing power.

Specific hardware circuits overcome speed constraints but are not compatible with a short time-to-market. They also need early and evaluative demonstration prototypes. An alternative can be provided by programmable software components like Digital Signal Processor (DSP) and Reduced Instruction Set Computer (RISC) or programmable hardware components like Field-Programmable Gate Array (FPGA). However, only one embedded processor is usually not sufficient for modern complicated applications.

As the complexity of the digital signal processing applications increases, multiple processing units are necessary in an embedded system to satisfy the requirement of great computation ability. An embedded system with several cores (e.g. multi-core DSP from TI [(1)]) and/or several hardware accelerators (e.g. IPs for Intellectual

(1). http://www.ti.com/

Properties) becomes a parallel embedded system. Hard real-time constraint must be satisfied by the multicomponent architecture, and the design can provide considerable flexibility since DSP, RISC and FPGA are programmable. However, for an embedded system with several processing cores, the flexibility is limited by the system's fixed topological structure and the fixed number of the programmable components.

More recent FPGAs offer very dense integration. With the help of multi-million gate configurable logic and various heterogeneous FPGA hardware components (multipliers, memory blocks, etc.), soft and hard processors could now be integrated on FPGA. Examples of such soft RISC processors include Nios from Altera (2) and MicroBlaze from Xilinx (3). In addition, Xilinx has also integrated the PowerPC 405 hard core on its FPGA. With multiple processors integrated on FPGA, we can build up FPGA based Multiprocessor System-on-Chip (MPSoC). As an example of the research for multiprocessor system, ATLAS is developed in the RAMP project (4) and is a prototype including 9 PowerPC 405 cores where 8 cores run multithreaded code for applications and the 9th core handles the operation system and I/O devices. Another example is the SocLib project (5) that addresses MPSoC.

MPSoC offers flexibility and efficiency, not only as regards its software but also as regards its hardware. Designers can directly elaborate several scenarios for architecture and algorithmic design exploring to reach real-time constraints, and the time-to-market is shorter in comparison with specific hardware circuits. However, such "manual explorations" are still complex, needing strong expertise and resulting in important time development. Therefore, "automatic solutions" with rapid prototyping methodologies are necessary for developing MPSoC.

Objective

Applications like digital signal processing usually consist of a set of tasks that are computations and communications between computations. Task scheduling is necessary when implementing such an application in a parallel computer system. Task scheduling consists in assigning and ordering computations and communications respectively to processors and communication links of the target system in order to finish all the tasks as soon as possible. The general task scheduling problem has been

(2). http://www.altera.com
(3). http://www.xilinx.com
(4). http://ramp.eecs.berkeley.edu/
(5). https://www.soclib.fr/

proven to be NP-hard, therefore, many works try to find heuristics for approaching the optimal solution. Early scheduling heuristics do not take communication into account. As the communication increases in modern applications, many heuristics now consider communication for task scheduling, and most of them use fully connected topology network in which all communications can be performed concurrently. Arbitrary processor networks are then used to accurately describe parallel computer systems, where communication links are not contention-free, and task scheduling takes communication contention into account.

This thesis mainly concerns the task scheduling problem in rapid prototyping for parallel embedded systems (e.g. MPSoC). We aim at task scheduling models for parallel embedded systems by accurately considering communications between computations. We also propose list scheduling heuristics with advanced techniques to improve the scheduling performance. Our scheduling methods are integrated in PREESM (Parallel & Real-time Embedded Executives Scheduling Method) (6) that is an Eclipse-based open source rapid prototyping framework. Scheduling is an important step in PREESM. The schedule result of an application on a parallel embedded system with our advanced heuristics will be further used to generate the code and finally to efficiently implement the application on the parallel embedded system.

Organization

After being briefly introduced, the rest of this work is organized in 5 chapters as follows.

Chapter 1 presents the rapid prototyping and hardware/software co-design problems. We firstly present the AAA (Adequation Algorithm Architecture) rapid prototyping methodology used for FPGA based multiprocessor system. Two different tools for generating Hardware Description Language (HDL) code from high-level languages are also presented. We introduce our rapid prototyping framework for hardware/software co-design at the end of this chapter.

Chapter 2 presents necessary graph models for rapid prototyping and hardware/-software co-design. We present several dataflow graphs to describe an application algorithm for parallel programming. As for task scheduling, the Directed Acyclic Graph (DAG) model is used to describe the application algorithm. Properties of the DAG model are also presented in details. The target system usually has the

(6). http://sourceforge.net/projects/preesm/

distributed memory architecture for parallel embedded systems. We propose a graph-based advanced architecture model to accurately describe parallel embedded systems. The advanced architecture model can be specified in a graph editor by respecting the IP-XACT standard.

Chapter 3 presents the task scheduling problem in parallel embedded systems. After a survey of the general task scheduling, we present the task scheduling with our advanced architecture model in detail. Since it is difficult to implement the task scheduling with the advanced architecture model at the very start, the advanced architecture model is slightly simplified to be the topology graph model. The scheduling problem with this simplified architecture model is the task scheduling with communication contention, and we will propose advanced techniques of this simplified scheduling for parallel embedded systems.

Chapter 4 presents list scheduling heuristics for the topology graph model (simplified architecture model in Chapter 3). We propose three new groups of node levels with communication contention that are used for generating static and dynamic node lists. With the addition of the other two existing groups of node levels, these five groups are all used in a list scheduling heuristic, which gives a combined heuristic. Experimental results are given at the end of this chapter to show the improvement of the scheduling performance. The time complexity of the list scheduling heuristic is also analyzed and tested to show its rapidity.

Chapter 5 gives two advanced techniques respectively named the critical child and the communication delay to extend the list scheduling heuristics of Chapter 4. These two techniques are combined with the five groups of node levels to further improve the scheduling performance. Though the time complexity increases by a factor of the number of processors, our experimental results show that the scheduling performance are greatly improved and the time complexity is acceptable.

We conclude this work in the end and also give some prospects.

1

Rapid Prototyping and Hardware/Software Co-design

1.1 Introduction

The multicomponent architecture of MPSoC [Mar06] raises problems in terms of application distribution: manual data transfers and synchronizations quickly become very complex and result in loss of time and potential deadlocks.

One suitable design process solution consists in using rapid prototyping methodology. The aim is then to go from a high-level description of the application to its real-time implementation on target architecture as automatically as possible. This automation saves development time and prevents conflicts and deadlocks. It ensures processing safety and reduces validation tests.

Hardware/software co-design [MG97] has been proposed as the design method for embedded systems and is used for designing System-on-Chip (SoC) [SBB06, OH06]. It needs to generate code for hardware coprocessors in Hardware Description Language (HDL) that is usually more complicated than software code. A tendency is to generate HDL code from high-level languages. This topic is interesting and important for hardware/software co-design.

This chapter presents rapid prototyping and hardware/software co-design. The rest of the chapter is organized as follows: Section 1.2 presents the AAA rapid prototyping for FPGA based MPSoC, and we use this rapid prototyping methodology

for designing multi-MicroBlaze systems on FPGA with the SynDEx tool. Section 1.3 presents two tools to generate HDL code from high-level languages for hardware/software co-design. When rapid prototyping is used with hardware/software co-design, we will need a new framework. Our new rapid prototyping framework is introduced in Section 1.4. The conclusion is given in Section 1.5.

1.2 Design FPGA based MPSoC with AAA Rapid Prototyping Methodology

The AAA (Adequation Algorithm Architecture) rapid prototyping methodology is suitable for designing image processing systems with heterogeneous multicomponent architectures. Based on this methodology, the SynDEx [(1)] tool has been used in some multi-DSP systems for image processing applications [RUN+05b]. This section presents the use of this rapid prototyping methodology in multi-MicroBlaze systems on FPGA. An extension of SynDEx is also introduced.

1.2.1 AAA Rapid Prototyping Methodology and SynDEx

SynDEx is a free academic system level Computer Aided Design tool developed by INRIA Rocquencourt. It supports the AAA methodology [GLS99] for distributed real-time processing. The aim of SynDEx is to directly achieve optimized implementation from descriptions of an algorithm and an architecture.

AAA in SynDEx

Figure 1.1 gives the SynDEx design flow. An algorithm graph is described as a Dataflow Graph (DFG), and specifies the potential parallelism of the application. An architecture graph describes the multicomponent target, i.e. a set of interconnected processors and specific integrated circuits, and specifies the available parallelism. In the application example given in Figure 1.1, the algorithm graph includes one input, two outputs and a function that is divided into two identical parts to be executed simultaneously. The architecture graph of the target system is composed of one PC, two MicroBlazes and two communication media. Since these two MicroBlazes and the medium between them are all integrated on an FPGA, the architecture graph gives

(1). http://www.syndex.org

a medium-coarse grain description in comparison with the one considering an FPGA as a black-box [RUN+05b].

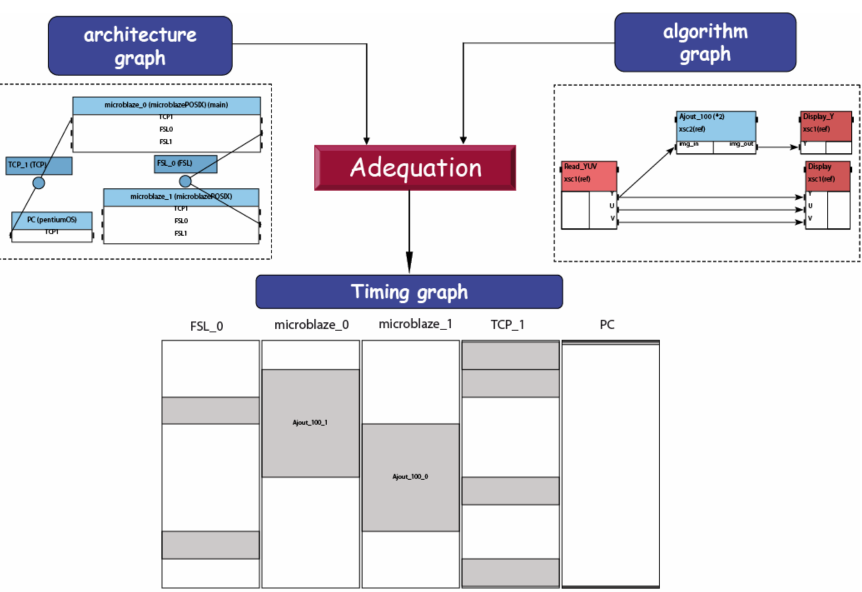


Figure 1.1: *SynDEx design flow*

"Adequation" (Figure 1.1) means efficient mapping, and consists in manually or automatically exploring the space of implementation solutions with optimization heuristics [GLS99]. These heuristics aim to minimize the total execution time of the algorithm running on the multicomponent architecture. The heuristic is a greedy list scheduling based approach with manual interaction when timing constraints are not met.

Implementation consists of both performing a distribution (allocating parts of the algorithm to components) and scheduling the algorithm on the architecture (i.e. giving a total order for the operations distributed onto a component).

Formal verifications during "adequation" avoid deadlocks in the communication scheme thanks to semaphores inserted automatically during real-time code generation. Moreover, since the Synchronized Distributed Executives are automatically generated and safe, part of the tests and low-level manual coding are eliminated, decreasing the development lifecycle.

SynDEx provides a timing graph (Figure 1.1), which includes simulation results of the distributed application and thus enables SynDEx to be used as a virtual prototyping tool. SynDEx then automatically generates the generic executives, which are independent of the hardware target, and places them in several source files, one for

each hardware target.

Automatic Executive Generation

The generic executives automatically generated by SynDEx are static and composed of a list of macro-calls. The M4 [(2)] macroprocessor transforms this list of macro-calls into compilable codes for a specific target. The codes are usually C or assemble codes for processors and VHDL for the specific functions implemented on the FPGA. The M4 macroprocessor replaces macro-calls by their definitions as given in the corresponding executive kernel. The definitions are dependent on a target and/or a communication medium. In this way, SynDEx can be seen as an off-line static operating system that is suitable for setting data-driven scheduling, such as image processing applications. For examples, SynDEx kernels have been developed for several processors such as General Purpose Processors (usually on PC), TMS320C6x (C62x, C64x) DSP and Virtex FPGA families [RUN+05b]. The generated codes could then be compiled by specific Computer-aided design (CAD) tools such as CCS for DSP, Quartus or ISE for FPGA and Visual Studio for PC.

1.2.2 Rapid Prototyping for Multi-MicroBlaze Systems on FPGA

The AAA rapid prototyping methodology has been used in a number of multi-DSP systems for image processing applications, and it could also be used in FPGA-based MPSoC to integrate multiple components on one or more FPGAs. As an embedded soft core, MicroBlaze is a RISC processor and optimized for implementation on Xilinx FPGA. It is highly configurable, allowing users to select a specific set of features required by their design. Integrating multiple MicroBlazes on one or more FPGAs can build up a multi-MicroBlaze MPSoC. This multi-MicroBlaze system is flexible in terms of both software and hardware, so it can be used in complicated and computation-rich applications such as image processing. This section details the use of rapid prototyping for multi-MicroBlaze systems on FPGA with SynDEx.

Design Flow

Figure 1.2 shows the design flow for multi-MicroBlaze systems. A number of tools such as SynDEx, M4, Embedded Development Kit (EDK) and Visual Studio are used

(2). http://www.gnu.org/software/m4

for different design stages.

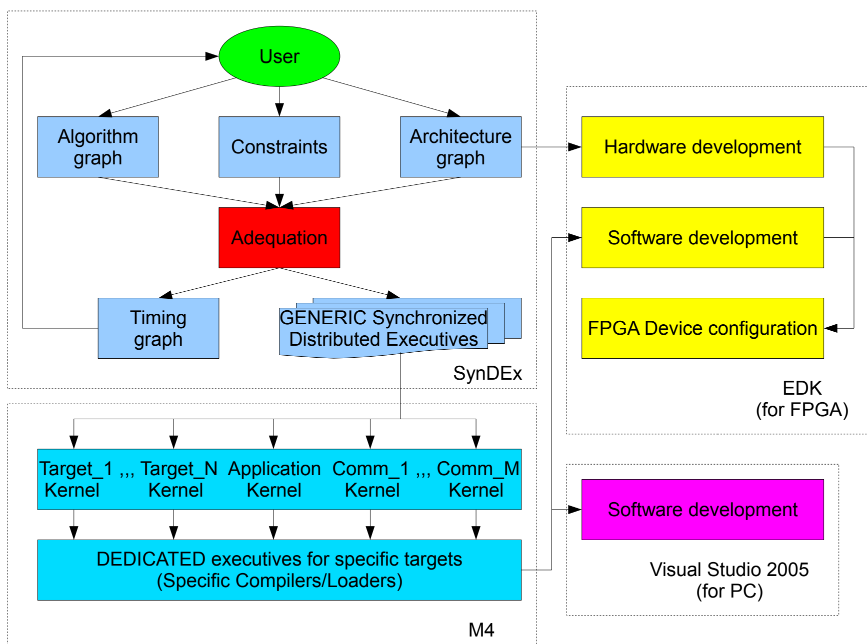


Figure 1.2: *Rapid prototyping design flow with SynDEx*

In this design flow, users firstly have to model in SynDEx Integrated Development Environment (IDE) the processors and communication media which are used in their design. Two different models are possible for the communication media between processors: Single Access Memory (SAM) and Random Access Memory (RAM, shared memory) [RUN+05b]. These modules are saved in a library and could be used for other designs without any modifications. With these modules, users then could build up the architecture and the algorithm graphs, and the "adequation" would be done by SynDEx while the Synchronized Distributed Executives are automatically generated in the form of m4 files. The m4 files then are translated into compilable executives for specific targets such as MicroBlaze, PC and specific functions on FPGA with the help of kernels which are explained in this section.

As the codes are generated, the next step is to build up the system. For the Xilinx FPGA, EDK is used for both hardware and software developments. The hardware is equivalent to the description of FPGA in the architecture graph of SynDEx except that it is described in the finest grain with EDK and can be used to generate the bitstream that configures the FPGA. For software programming, EDK uses the generated executives for MicroBlazes and respective drivers to build up Executable Linked Format (ELF) files for the multi-MicroBlaze system. When PC is used, Vi-

sual Studio is necessary for software development using the generated executives and several drivers for PC.

SynDEx Executive Kernels

As described in Section 1.2.1, the SynDEx generic executives have to be translated into a compilable language. The translation of SynDEx macros into the target language is contained in library files (also called kernels). Figure 1.3 shows the organization of different kernels for the multi-MicroBlaze system. There are two types of processors (PC and MicroBlaze) and two types of communication media (Fast Simplex Link (FSL) and TCP/IP) in the multi-MicroBlaze system. FSL is used for the communication between MicroBlazes, and TCP/IP is used for the communication between MicroBlaze and PC. The kernel for PC has been used in [RUN+05a], and the following explains the new kernels for MicroBlaze, FSL and TCP/IP.

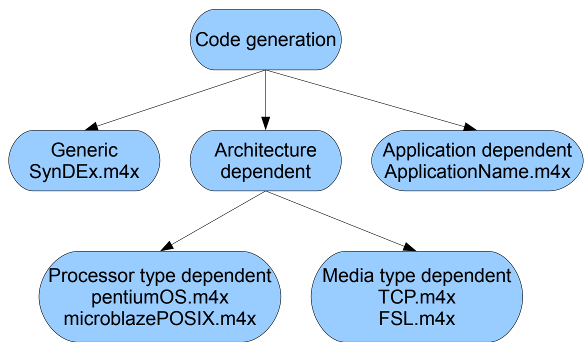


Figure 1.3: *SynDEx kernel organization*

MicroBlaze Kernel

The program in MicroBlaze-based systems could be designed using either a standalone Board Support Package (BSP), which has no operating system, or Xilkernel, which supports the core features required in an embedded real-time operating system (RTOS). Xilkernel is a POSIX compliant API. When using Xilkernel, the standalone BSP is used below the operating system layer. In [RRND06], RTOS is introduced in the AAA methodology. The RTOS has an impact on processor target such as execution time or allocated memory, but the over-cost is slight, especially for image processing algorithms where data are often large. Moreover, executives automatically generated including RTOS primitives are simple leading to a better comprehension for

users. They are also more generic and compatible with more components. Therefore, our software component kernel for MicroBlaze has been developed using Xilkernel.

A software component kernel is used to automatically generate executives that would run in a specific processor, and different kernels should be used for different processors. With the MicroBlaze kernel, the generic executives generated by SynDEx are translated into MicroBlaze compilable C codes. The generated codes are compiled using Xilkernel, and semaphores are used to synchronize the various threads of the program.

Executives generated by SynDEx consist of a sequential list of function calls (one for each DFG operation). Therefore, functions have to be defined outside of SynDEx to make the whole program executable. Most of these functions are developed in C language so that they can be reused for any C programmable device.

Communication Media Kernels for Multi-MicroBlaze Systems

FSL: FSL is a uni-directional point-to-point communication channel bus used to provide fast communication between two IPs. Since FSL is a First-In-First-Out (FIFO) based communication bus, the kernel is developed based on the SAM model in SynDEx. C functions are developed for MicroBlaze to send/receive data to/from FSL, and the calling of these functions is automatically generated for MicroBlazes.

TCP/IP: TCP/IP could be modeled as a SAM in SynDEx because it uses FIFOs. With the kernel developed for TCP/IP, SynDEx could generate a sequence of generic executives to complete TCP/IP-based communication. Like the computation function requirements, the communication functions should also be developed outside SynDEx, and these functions may be different depending on the different types of processors. C functions have been respectively developed for PC and MicroBlaze so that they can communicate using TCP/IP.

Comments on SynDEx

The main advantage of the SynDEx based prototyping process is its simplicity because most of the tasks performed by the user concern the description of an application (creation of the algorithm graph) and a compiling environment. All complex tasks (adequation, synchronization, data transfers and chronometric reports) are executed automatically or semi-automatically. The user can rapidly explore several

design alternatives by modifying the architecture graph and/or the algorithm graph, or by adding constraints.

There are some disadvantages in SynDEx. The two models of SAM and RAM are not suitable to accurately describe the actual advanced communication media like switch-base network or Network-on-Chip (NoC). SynDEx is usually used for software code generation in multiprocessor systems; however, it is not natural to use SynDEx in a heterogeneous system with processors and IP coprocessors. Since the hardware code generation for coprocessors is usually more complicated than the software code generation for processors, the AAA rapid prototyping methodology is extended to be used for hardware code generation in another tool called SynDEx-Ic. This tool is briefly presented in the next subsection.

1.2.3 SynDEx-Ic Tool

SynDEx-Ic[3] is a free software developed by the "Conception d'architecture" group of A2SI laboratory in ESIEE. It is a rapid prototyping software for real time applications as an extension of SynDEx. In comparison with SynDEx, SynDEx-Ic covers the architectures based on dedicated circuits of Application-Specific Integrated Circuit (ASIC) and/or FPGA and generates the synthesizable VHDL code.

SynDEx-Ic extends the AAA rapid prototyping methodology to the hardware implementation of real-time applications onto specific integrated circuits [KASG03]. The algorithm is modeled by a Factorized Data Dependence Graph (FDDG) and is specified by using the tool's graphical interface. The objective is to find an offline (i.e. before the execution) implementation of the algorithm to meet a given latency constraint (execution time). This implementation also tries to minimize the required hardware resources on the target circuit (for example the number of Configurable Logical Blocks for FPGA). If an implementation of the factorized specification does not meet the real time constraints, it is necessary to defactorize the implementation graph. The more a graph is defactorized, the greater the parallelism is, and the more it is possible to reduce the latency. However, the use of resource increases when the graph is defactorized.

The optimized implementation of a factorized algorithm graph onto the target architecture is formalized in terms of graph defactorization transformation. This optimization problem is known to be NP-hard [GJ79], and its size is usually huge

(3). http://www.esiee.fr/ grandpit/web-ca/syndex-ic/index.htm

for realistic applications. SynDEx-Ic uses a heuristic based on a greedy algorithm coupled with simulated annealing heuristics. The result of the heuristic is then directly converted into synthesizable VHDL code to run on the target component or to be simulated. Figure 1.4 shows the design flow of SynDEx-Ic.

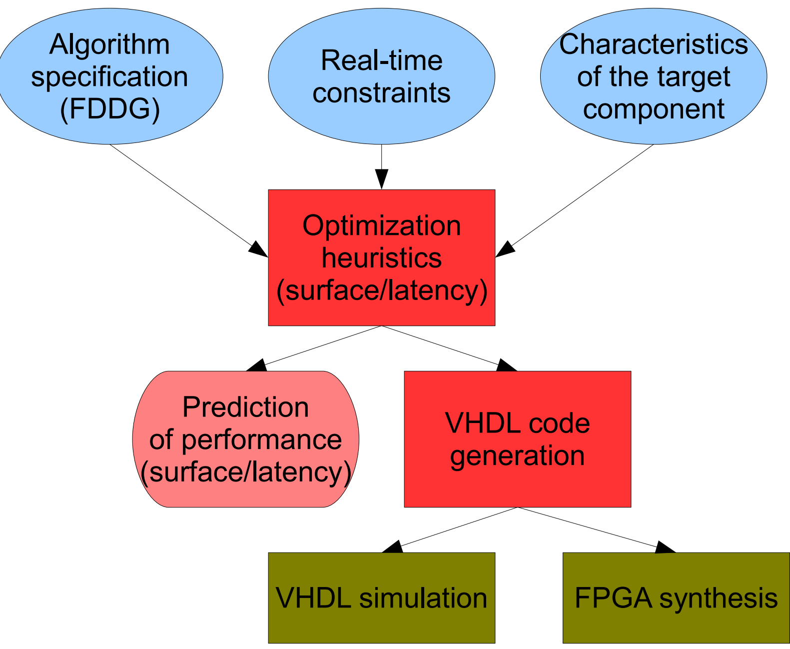


Figure 1.4: *SynDEx-Ic design flow*

SynDEx-Ic needs some libraries to generate the final VHDL code. In addition to the libraries included in the tool, users have to create an application library to define some specific operations by using VHDL. Since the hardware programming in VHDL is usually more complicated than the software programming like in C, we need to search other simpler ways to generate hardware code for hardware/software co-design.

1.3 Tools for HDL Code Generation

Hardware/software co-design usually needs both hardware programming for Intellectual Property (IP) coprocessors and software programming for general processors, and the hardware programming is usually more complicated than the software programming. However, it is possible to generating HDL code from high-level language like C. This section introduces two tools for generating HDL code from high-level languages.

1.3.1 GAUT: A High-Level Synthesis Tool

GAUT (Génération Automatique d'Unités de Traitement) [(4)] is a high-level synthesis tool developed in Lab-STICC in Lorient (France). It dedicates to Digital Signal Processing DSP applications from an algorithmic specification in C [LGCH+05]. GAUT generates an IEEE P1076 compliant Register Transfer Level (RTL) VHDL file. This file is an input for commercial, off-the-shelf, logical synthesis tools like ISE from Xilinx, Design Compiler from Synopsys, Quartus from Altera, ...

GAUT Design Flow

Figure 1.5 shows the design flow of GAUT. Starting from a pure C function GAUT extracts the potential parallelism before selecting/allocating operators, scheduling and binding operations. GAUT synthesizes a potentially pipelined architecture composed of a processing unit, a memory unit, a communication and multiplexing unit and a GALS/LIS interface [BMB05, CCB+05, LGCHM05, CSB+04].

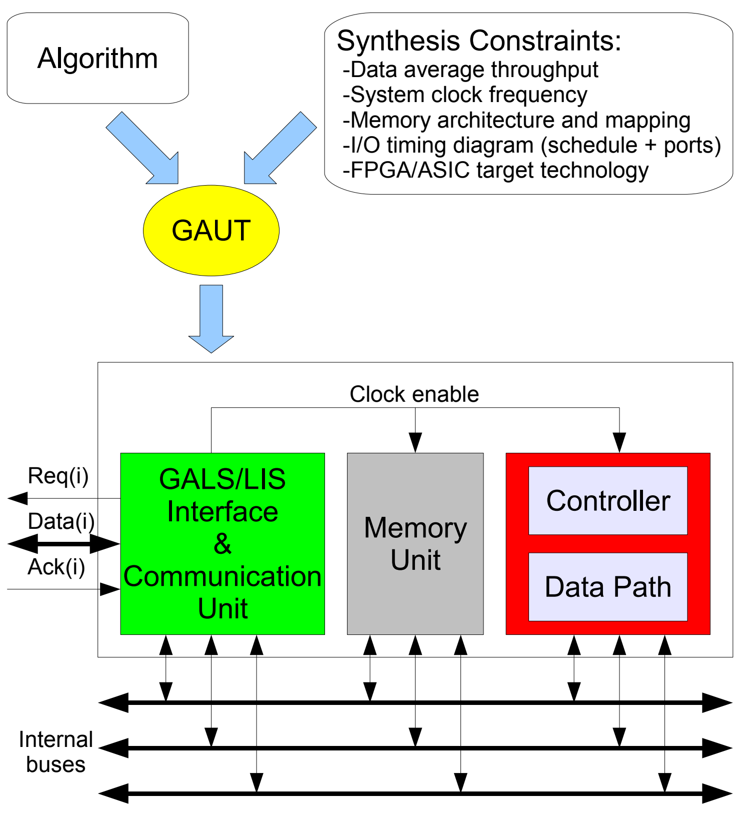


Figure 1.5: *GAUT design flow*

(4). http://www-labsticc.univ-ubs.fr/www-gaut/

Use of GAUT

The tool structure in GAUT is shown in Figure 1.6. It mainly consists of seven blocs which are explained as follows:

A: Algorithm analysis, extraction and visualization of parallelism;

B: Management of libraries;

C: Generation of Memory Unit (memory banks and associated controllers);

D: Synthesis of architecture under constraints of sampling rate;

E: Generation of the communication/protocol interface (FIFO protocol, LIS interface, ...);

F: Visualization of the Gantt chart of the scheduled operations, IOs and memory access;

G: Generation of the testbench and simulation with ModelSim.

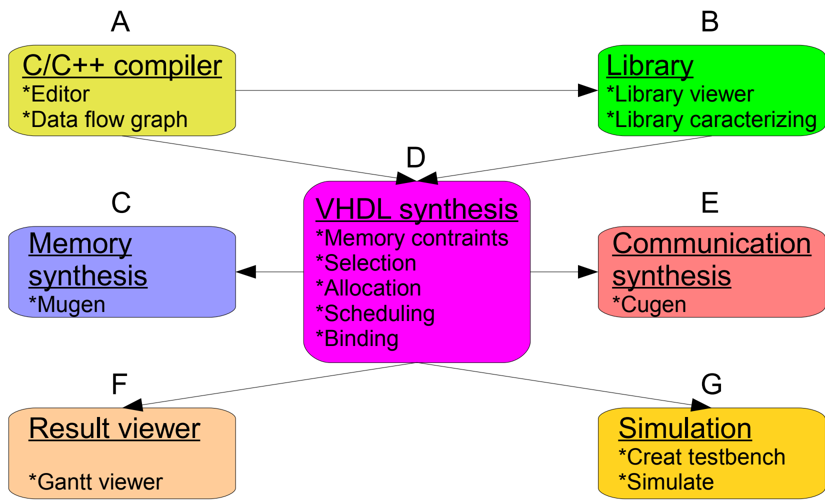


Figure 1.6: *GAUT tool structure*

The C or C++ code is developed and compiled in the editor window. GAUT compilation is a parallelism extraction that allows the creation of the dependency graph file necessary for the VHDL generation. Depending on algorithm specified before the compilation the generated graph contains the necessary computing operators, inputs, outputs and buffers. To generate the VHDL code, GAUT requires specifying a cadency, a clock frequency and the type of architecture (with or without memory unit). The VHDL generation depends on the dependency graph and the chosen library. After VHDL generation, the VHDL testbench can be automatically generated by the

tool. A direct connection between GAUT and ModelSim is possible just by specifying the ModelSim path.

Limits of GAUT

The use of GAUT to transform the C code of an image codec presents some limits of this tool. In fact, GAUT can not deal with very complex C codes. After achieving some tests we noted the following limits:

- GAUT does not support dynamic structures: The C code in the input of the tool must be deterministic which means that all the variable sizes and the treatment length have to be known by the tool. Consequently, it is not possible to compile algorithms containing pointers or while or switch structures.
- Function appeal: We cannot use predefined functions so all functions in the input of the tool must be main ones.
- Preprocessor directives: Some codes contain preprocessor directives used for a DSP compilation. These directives can not be compiled with GAUT, so they have to be eliminated.
- Graphs containing a huge number of knots: If the treatment is very complex and requires a dependency graph with a huge number of knots, the graph generation can not be achieved, and we would not have the VHDL code.

The results obtained by GAUT were not satisfying. The tool, in the 2.2.0 version, was considerably limited to treat complex codes. Even the functions that we succeeded to transform into RTL level VHDL code presented some synthesis problems in Xilinx ISE tool. Some of the synthesized functions contained a big number of states in their FSM and consumed a huge percentage of the FPGA area. Considering these results, we decide to adopt another method for the automatic transformation using the CAL language.

1.3.2 Open Dataflow Framework

Open Dataflow (OpenDF [5] for short) is an environment for building and executing actor/dataflow models, including support for the CAL actor language. It is also a compilation framework that consists of tools to compile CAL to HDL(VHDL/Verilog) for hardware implementation [JMP+08] and to C for integration with the SystemC tool chain [RWR+08]. Work on mixed HW/SW implementations is under way.

(5). http://opendf.sourceforge.net/

CAL Language

CAL [EJ03, Jan07] is a domain-specific language that provides useful abstractions for dataflow programming with actors. CAL has been used in a wide variety of applications [LMTJ07]. This section gives a brief introduction to the key elements of *actor* and *network* for the CAL language.

Actor: An actor is a parametric entity with inputs, outputs and an internal state. An actor can not change the state of another actor in the network, but it can communicate with others by exchanging tokens through connected inputs/outputs. The execution of an actor is based on the execution of elementary functions called actions. The modeling of the actor states can be done using a finite state machine with the appropriate priorities if necessary.

While executing an action some tokens are consumed, and others are produced independently from the current state of the actor. The execution of an action can be controlled by a finite state machine or by a specified condition using the "guard" syntax or both of them. The "guard" is an expression to test the value of an input token or a local variable. If more than one action can be executed at the same time, it is very important to define the priority between them. Therefore, the notion of priority has been introduced in the language. This notion is very important for the finite state machines in case of concurrent actions. The actor functioning can be scheduled using a finite state machine. The required informations are the initial state and the action that changes the current state to the next state.

The basic structure of a CAL actor is shown in the `Add` actor below, which has two input ports `t1` and `t2`, and one output port `s`, all of type `T`. The actor contains one *action* that consumes one token on each input ports, and produces one token on the output port. An action may *fire* if the availability of tokens on the input ports matches the *port patterns*, which in this example corresponds to one token on both ports `t1` and `t2`.

```
actor Add() T t1, T t2 ⇒ T s :
  action [a], [b] ⇒ [sum]
  do
    sum := a + b;
  end
end
```

Network: A set of CAL actors are instantiated and connected to form a CAL application, i.e. a CAL network.

Figure 1.7 shows a simple CAL network `Sum`, which consists of the previously defined `Add` actor and a delay actor `Z`. The network itself has input and output ports, and the instantiated entities may be either actors or other networks, which allows for a hierarchical design. The source code for the delay actor `Z` and the network `Sum` is found below as well as the XML description of the network.

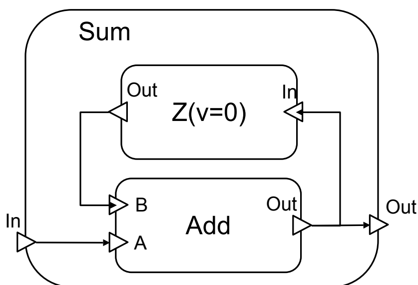


Figure 1.7: *A simple* CAL *network*

```
actor Z (v) In ⇒ Out:

  A: action ⇒ [v] end
  B: action [x] ⇒ [x] end

  schedule fsm s0:
    s0 (A) −−> s1;
    s1 (B) −−> s1;
  end
end
```

```
network Sum () In ⇒ Out:

entities
  add = Add();
  z = Z(v=0);

structure
  In −−> add.A;
  z.Out −−> add.B;
  add.Out −−> z.In;
  add.Out −− > Out;
end
```

```
<?xml version="1.0" encoding="UTF-8"?>
<XDF name="Sum">
   <Port kind="Input" name="In"/>
   <Port kind="Output" name="Out"/>
   <Instance id="add"/>
   <Instance id="z">
      <Class name="Z"/>
      <Parameter name="v">
         <Expr kind="Literal" literal-kind="Integer" value="0"/>
      </Parameter>
   </Instance>
   <Connection dst="add" dst-port="A" src="" src-port="In"/>
   <Connection dst="add" dst-port="B" src="z" src-port="Out"/>
   <Connection dst="z" dst-port="In" src="add" src-port="Out"/>
   <Connection dst="" dst-port="Out" src="add" src-port="Out"/>
</XDF>
```

Formerly, networks have been traditionally described in a textual language, which can be automatically converted to FNL (Functional unit Network Language, XML language standardized in RVC [LMTJ07]) and vice versa. Graphiti editor, which is presented in Section 1.4.1, is available to create, edit, save and display a network.

Hardware Synthesis - CAL2HDL

CAL program must be implemented in real systems. Therefore, it should be translated to other technique languages for hardware and software synthesis, and the translation should be automatic. In fact, OpenDF is also a compilation framework, and there are backends for converting CAL program to HDL and C programs.

CAL program is translated to HDL by using a tool named CAL2HDL [JMP+08]. When generating hardware implementations from a network of CAL actors, each actor is separately translated to RTL description in Verilog, and a number of actor instances that are references of the generated RTL descriptions are connected with FIFOs to elaborate the network structure in VHDL. Figure 1.8 shows the CAL2HDL tool structure in OpenDF [Xil08].

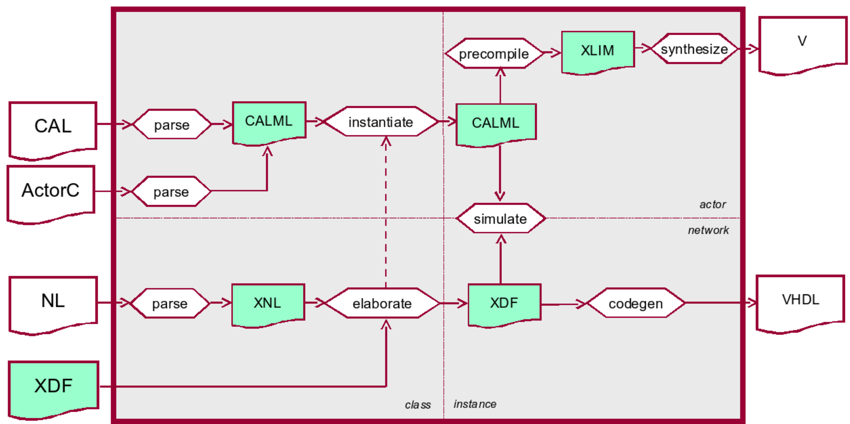


Figure 1.8: *CAL2HDL tool structure in OpenDF*

CAL Simulation

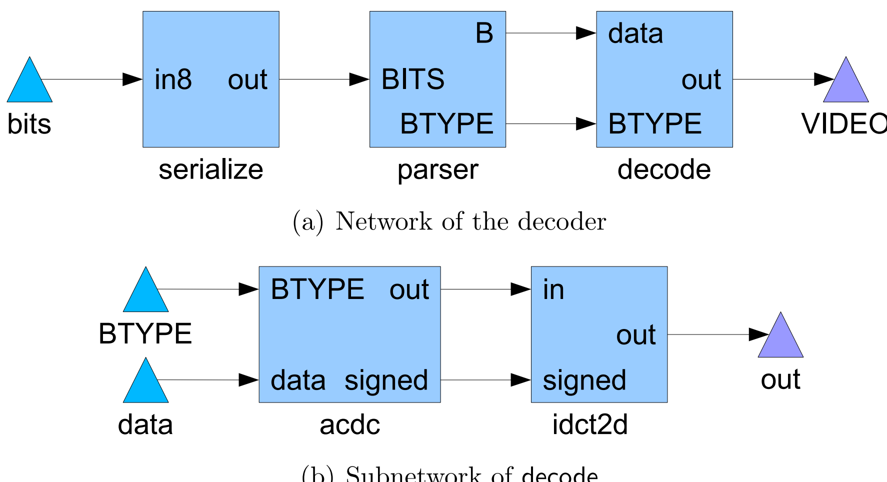
CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses (6) project. Moses features a graphical network editor, and allows the user to monitor actors'

(6). http://www.tik.ee.ethz.ch/moses/

execution (actor state and token values). The project being no longer maintained, it has been superseded by an Eclipse environment composed of 2 tools/plugins - the OpenDF environment for CAL editing and the Graphiti editor for graphically editing the network.

An Design Example with OpenDF

We give an example of implementing an MPEG-4 Part 2 decoder for intra frames on a FPGA platform from Xilinx. The decoder is expressed as a CAL algorithm, and the top level network is shown in Figure 1.9(a). The decoder consists of one input port, three entities and one output port. The serialize is an entity of actor; the parser is an entity of subnetwork composed of actors; and the decode is an entity of subnetwork composed of other two subnetworks. Figure 1.9(b) shows the two subnetworks acdc and idct2d of the entity decode.



(a) Network of the decoder

(b) Subnetwork of decode

Figure 1.9: *Algorithm of MPEG-4 Part 2 decoder*

Figure 1.10 shows the architecture of a system containing a computer and an ML402 platform [(7)] of Xilinx. This system uses the FPGA and the external memory of the ML402 platform. The FPGA contains a processor of MicroBlaze (uB), an IP (IP1) connected to the MicroBlaze by two FIFOs (FSL1 and FSL2) of FSL, and another IP (IP2) connected to the MicroBlaze by the bus of PLB (Processor Local Bus). The external memory (M) is connected to the MicroBlaze by the bus PLB, and the computer (PC) is connected to the MicroBlaze by the bus TCP with TCP/IP protocol.

(7). http://www.xilinx.com/products/devkits/HW-V4-ML402-UNI-G.htm

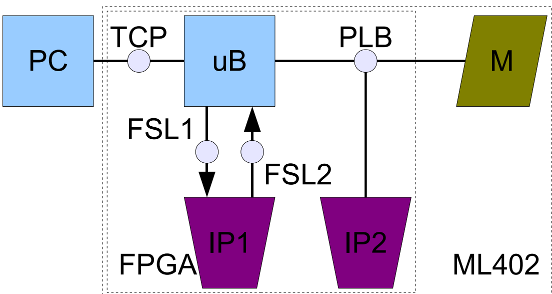


Figure 1.10: *Architecture of PC+ML402*

The MPEG-4 Part 2 decoder is implemented on the architecture of PC+ML402. The top level network is firstly compiled to threads of SystemC by CAL2C [RWR+08], then it is manually translated to be executed on MicroBlaze with a POSIX API targeting embedded kernel. The two entities serialize and parser are executed on the MicroBlaze by calling functions, and the entity decode is executed on the coprocessor IP1. IP1 is generated by CAL2HDL [JMP+08], and an additional interface is manually created and added between IP1 and the two FSLs. The output VIDEO is transferred from IP1 to the external memory M on the platform by the MicroBlaze. VIDEO is finally displayed by IP2, which is a hardware display controller IP core of Xilinx. All these entities are synchronized on the MicroBlaze by using semaphores.

1.3.3 Comparison between GAUT and OpenDF

Though GAUT and OpenDF can both generate HDL code from high level languages, they differ from each other in some aspects. A comparison of these two tools is given as follows:

Criteria	GAUT	OpenDF
General use	The initial C code has to be simple, deterministic and with a minimum of control structures. The perfect code would be a computing algorithm as matrix product, transforms (Fourier, Laplace, ...), interpolation, ...	CAL can be used for both simple and complex structures. Being a high level language, the code development is relatively simple.

HDL code generation	GAUT offers the possibility to choose the aspect of the final structures. If a C code is correctly compiled, the generated VHDL code is synthesizable and correctly simulated.	The Generated codes are a VHDL one for the top file and a Verilog one for each actor. Connections, inputs and outputs are transformed into FIFOs synchronized by consumption of tokens.
Compilation	While compiling a C/C++ code, GAUT extracts the parallelism structures and creates a *.cdfg file that allows the dependency graph generation. This graph is used for the VHDL code generation.	The CAL2HDL compilation is a generation of intermediate files that enable the HDL code generation. These files are *.xlim, *.sxlim, *.ssacalml and *.pcalml.
Simulation	The generated VHDL of GAUT can be directly simulated via a connection between GAUT and ModelSim. An adequate testbench is automatically generated.	The CAL code can be simulated before the HDL code generation. But for hardware simulation, a testbench has to be written.
Limits	GAUT can not support dynamic structures, pointers and non-deterministic algorithms.	OpenDF can not generate HDL for repeat structures because an actor is unable to consume more than one token in an input at a precise time.

1.4 An Eclipse-Based Open Source Rapid Prototyping Framework

When we have the hardware/software codes for specific operations, we then need a tool to implement the rapid prototyping methodology for parallel embedded systems. This section introduces an Eclipse-based open source rapid prototyping framework. Figure 1.11 shows the framework structure. It is made up of three tools to

increase their reusability in different contexts. The three tools are Graphiti, SDF4J and PREEM; they are detailed as follows.

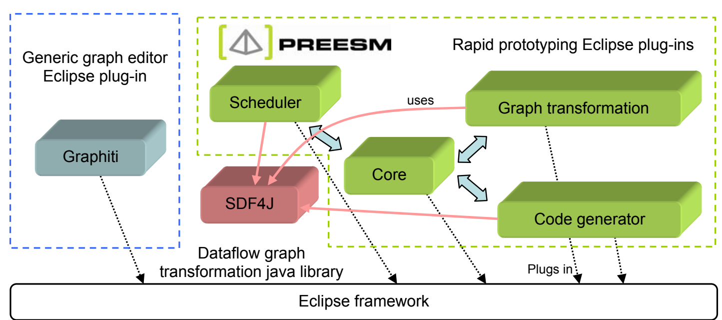


Figure 1.11: *An Eclipse-based rapid prototyping framework*

1.4.1 Graphiti: A Generic Graph Editor for Editing Architectures, Algorithms and Workflows

The first step of rapid prototyping is to describe the target algorithm and architecture graphs. A graphical editor decreases the development time to create, modify and edit those graphs. Graphiti [8] is an open-source plug-in for the Eclipse environment and is provided to support algorithm and architecture graphs for the proposed framework. Graphiti can also be quickly configured to support any type of file formats used for graph descriptions.

Graphiti is written using the Graphical Editor Framework (GEF). The editor is generic in the sense that any type of graph can be represented and edited. Graphiti is being used routinely with the following graph types and associated file formats: CAL networks (cf. Section 1.3.2), a subset of IP-XACT (cf. Section 2.3.3), GraphML (cf. Section 1.4.2) and PREESM workflows (cf. Section 1.4.3).

Overview of Graphiti

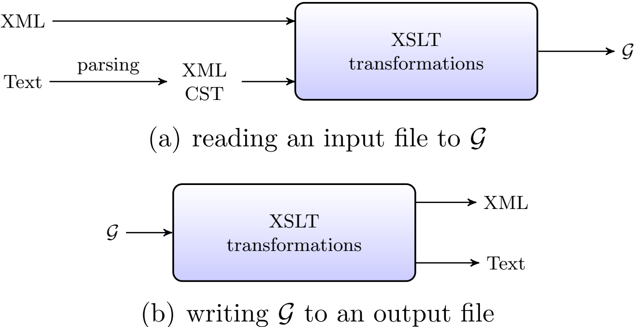
A type of graph is registered within the editor by a *configuration*. A configuration is an XML (Extensible Markup Language) file that describes:

1. the *abstract syntax* of the graph: types of vertices and edges, attributes allowed for objects of each type;
2. the *visual syntax* of the graph: colors, shapes, etc.;

(8). http://sourceforge.net/projects/graphiti-editor/

3. the transformations from the file format in which the graph is defined to the XML file format $\mathcal{G}$ of Graphiti, and vice-versa (Figure 1.12).

Two kinds of input transformations are supported, from XML to XML and from text to XML (Figure 1.12). Both these transformations are independent from the code of the editor: XML is transformed to XML with XSLT (Extensible Stylesheet Language Transformations), and text is parsed to its Concrete Syntax Tree (CST) represented in XML according to a LL(k) grammar by the Grammatica [(9)] parser. Similarly, two kinds of output transformations are supported, from XML to XML and from XML to text.



(a) reading an input file to $\mathcal{G}$

(b) writing $\mathcal{G}$ to an output file

Figure 1.12: *Input/output with Graphiti's XML format* $\mathcal{G}$

Graphiti handles *attributed graphs*. According to [JE01], an attributed graph is defined as a directed multigraph $G = (V, E, \mu)$ where

- V is the set of vertices;
- E is the multiset of edges (there can be more than one edge between any two vertices);
- μ is a function $\mu : (\{G\} \cup V \cup E) \times A \mapsto U$ that gives the attribute value (from the set of possible attribute values U) for an instance (from $\{G\} \cup V \cup E$)) with an attribute name (from the attribute name set A).

A built-in *type* attribute is defined so that each instance $i \in \{G\} \cup V \cup E$ has a type $t = \mu(i, \textbf{“type”})$, and only admits attributes from a set $A_t \subset A$. Additionally, a type t has a visual syntax: $\sigma(t)$ defines the color, shape and size associated with instances of type t.

Editing a graph with Graphiti is done as follows. The user selects a file and a set of matching configurations is computed based on the file extension. If the set contains more than one configuration, Graphiti asks the user which one is suitable for

(9). http://grammatica.percederberg.net/

the input file. The transformations defined in the configuration file are then applied to the input file, which results in a graph defined in Graphiti's XML format $\mathcal{G}$ as shown in Figure 1.12. The editor uses the visual syntax defined by σ in the configuration to draw the graph, vertices and edges. For each instance of type t the user can edit the relevant attributes allowed by $\tau(t)$ as defined in the configuration. Saving a graph consists in writing the graph in $\mathcal{G}$, and transforming it back to the input file's native format.

Editing a Configuration for a Graph Type

To create a configuration for the graph in Figure 1.13, a single type of vertex called "node" has to be defined. A "node" has an unique identifier called "id", and accepts a list of "values" initially equal to [0] (Figure 1.14). Additionally, ports need to be specified on the edges, so the configuration describes an `edgeType` element (Figure 1.15) that carries "sourcePort" and "targetPort" parameters to respectively store an edge's source and target ports, such as **acc**, **in**, and **out** in Figure 1.13.

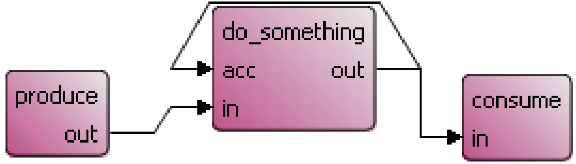


Figure 1.13: *A sample graph*

Graphiti is a tool totally independent of the PREESM tool. However, it generates workflows, IP-XACT and GraphML files that are the main inputs of PREESM. The GraphML files contain the algorithm description. They are loaded and stored in PREESM by the SDF4J library that is discussed in next section. The architecture description respects the IP-XACT standard. It is more specific than the algorithm description, and no external library is defined for architecture handling.

1.4.2 SDF4J: A Java Library for Algorithm Dataflow Graph Transformation

An algorithm is described as a Synchronous Dataflow Graph (SDF). The SDF model is a natural solution to describe algorithms with static behaviors [LM87b]. SDF4J [(10)] is an open source library providing usual transformations of SDF graphs

(10). http://sourceforge.net/projects/sdf4j/

```
<vertexType name="node">
   <attributes>
      <color red="163" green="0" blue="85"/>
      <shape name="roundedBox"/>
      <size width="40" height="40"/>
   </attributes>
   <parameters>
      <parameter name="id" type="java.lang.String" default=""/>
      <parameter name="values" type="java.util.List">
         <element value="0"/>
      </parameter>
   </parameters>
</vertexType>
```

Figure 1.14: *The type of vertices of the graph shown in Figure 1.13*

```
<edgeType name="edge">
   <attributes>
      <directed value="true"/>
   </attributes>
   <parameters>
      <parameter name="source␣port" type="java.lang.String" default=""/>
      <parameter name="target␣port" type="java.lang.String" default=""/>
   </parameters>
</edgeType>
```

Figure 1.15: *The type of edges of the graph shown in Figure 1.13*

in the Java programming language. SDF4J stands for Synchronous Dataflow For Java. This library aims at providing the user with a large choice of easily expandable Dataflow models associated to algorithm transformations and optimizations. The library also defines its own graph representation based on the GraphML [BEH+01] standard and provides the associated parser and exporter classes.

The SDF4J library defines several Dataflow graph models like SDF graph and Directed Acyclic Graph (DAG). It provides the user with several classic SDF transformations like hierarchy flattening, HSDF transformation and SDF to DAG transformation. These transformations are explained as follows:

- The *hierarchy flattening* aims at flattening the hierarchy (remove hierarchy levels) at the chosen depth in order to later extract as much as possible parallelism from the designer hierarchical description.
- The *HSDF transformation* transforms an SDF graph to an Homogeneous SDF graph in which the amount of tokens exchanged on edges are homogeneous (production = consumption). The HSDF model reveals all the potential parallelism of the application but dramatically increases the amount of vertices in the graph.
- The *SDF to DAG transformation* transforms an HSDF graph to a DAG which is commonly used for task scheduling.

1.4.3 PREESM: A Complete Framework for Hardware/Software Co-design

The PREESM [PRP+08] project performs the rapid prototyping tasks. PREESM uses the Graphiti and SDF4J tools to design algorithm and architecture graphs and generates their transformations. The PREESM core is an Eclipse plug-in that executes workflows. A workflow is a directed graph representing lists of rapid prototyping tasks to be executed with the input algorithm and architecture graphs. The rapid prototyping tasks are delegated to PREESM plug-ins. All these plug-ins are optional and appear as vertices in a workflow graph. PREESM is also built to be easily extensible with new plug-ins. There are three PREESM plug-ins at present: the graph transformation plug-in, the scheduler plug-in and the code-generation plug-in.

Figure 1.16 describes a classic workflow which can be applied in the PREESM tool. As seen in Section 1.4.2, the first dataflow model chosen to describe applications in PREESM is the SDF model. This model has the great advantage of enabling formal verification of static schedulability. The typical number of vertices to schedule in PREESM is between a hundred and a few thousands. The architecture is described based on the IP-XACT standard [SPI08] that is an IEEE standard from the SPIRIT consortium. The typical size of an architecture in PREESM is between a few cores and a few dozens of cores. A scenario is defined as a set of parameters and constraints that specify the conditions under which the deployment will run.

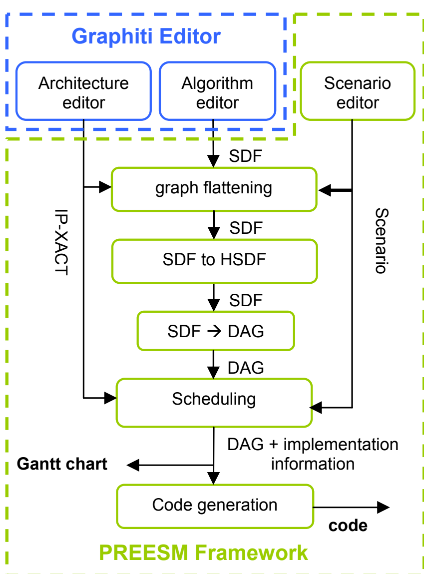


Figure 1.16: *A workflow graph: From SDF and IP-XACT descriptions to code*

Prior to entering in the scheduling phase, the algorithm goes through three steps of transformations: hierarchy flattening, HSDF transformation and SDF to DAG transformation. These transformations prepare the graph for the static scheduling and are provided by the Graph Transformation Module. Subsequently, the SDF graph converted into a Directed Acyclic Graph (DAG) is processed by the scheduler. As a result of the deployment by the scheduler, code is generated and a Gantt chart of the execution is displayed. The generated code consists of well scheduled function calls, synchronizations and data transfers between cores. The functions themselves are hand-written.

1.5 Conclusion

This chapter presented rapid prototyping and hardware/software co-design. AAA rapid prototyping methodology has been used in SynDEx to design multiprocessor embedded systems for many applications like digital signal processing and video compression. We used this methodology for multi-MicroBlaze systems on FPGA. Codes are automatically generated for each MicroBlaze with the developed kernels. Hardware/software co-design also needs to generate HDL code for hardware coprocessors that is usually more complicated than software code. GAUT and OpenDF are both tools for generating HDL code from high-level languages. The generated HDL code can be synthesized and implemented on FPGA as a complete design or a part of an embedded system.

We presented a new rapid prototyping framework of PREESM that supports hardware/software co-design for parallel embedded systems. PREESM firstly models an algorithm and an architecture as graphs, then it schedules the algorithm onto the architecture. The schedule results are finally used to generate code for the multiple processors of the architecture. We will mostly concern the scheduling problem in the following of this work. Therefore, the graph models used for scheduling will be explained in the next chapter. The scheduling problem will be deeply studied in the Chapter 3, 4 and 5.

2
Graph Models for Parallel Embedded Systems

2.1 Introduction

The recent evolution of embedded applications like digital communication and video compression has dramatically increased the algorithm and system complexities. In this work the application algorithm and the target system are respectively called algorithm and architecture. When a complicated algorithm is implemented on a parallel architecture for efficient computation, graphs [Die05] are usually used to model the algorithm and the architecture in order to facilitate the programming.

An algorithm can be modeled as different types of graph according to the different objectives. Dataflow graph is commonly used and consists in modeling an algorithm as a directed graph of data flowing between operations. Since an algorithm needs to be scheduled on the multiple processors of a parallel architecture, it is modeled as a Directed Acyclic Graph (DAG) for task scheduling, where nodes represent computations and edges represent communications between computations.

Computer architecture has changed from single-processor systems to parallel systems [Dun90, ERAEB05]. Parallel architectures are classified as the shared memory architecture and the distributed memory architecture. Parallel computation also needs appropriate models to describe architectures. The first model for the shared memory architecture was the parallel random access machine (PRAM) in [FW78].

Since PRAM is not accurate to describe a real parallel system, some other models were introduced to accurately describe real parallel systems. For example, the LogP model [CKP+93] uses four parameters to roughly describe the parallel system and is a balance between detail and simplicity.

The trend of parallel architecture is also coming in the domain of embedded systems. Parallel embedded systems usually use distributed memory architectures. For performance analysis and estimation, distributed memory architectures are usually modeled as completely connected graphs. However, this completely connected graph model is not yet accurate for parallel embedded systems. In [GS03], a parallel embedded system is modeled as an architecture graph containing four kinds of vertices corresponding to operator, communicator, memory and bus. However, this architecture graph does not describe advanced components like switches which are commonly used in parallel embedded systems for connecting multiple buses. Therefore, we need a new architecture model to appropriately describe parallel embedded systems.

This chapter introduces different existing algorithm and architecture models and proposes an advanced architecture model. The advanced architecture model will be used in the task scheduling for parallel embedded systems. This chapter is organized as follows: Section 2.2 introduces several graph models for algorithms. Then different architecture models are given in Section 2.3 including our advanced architecture model. The chapter is concluded in the end.

2.2 Algorithm Model

An algorithm can be modeled in different ways according to different objectives. These models are usually directed graphs, and this section presents some graph models for describing an algorithm.

2.2.1 Dataflow Model

Many applications such as signal processing applications aim to transform data. Systems implementing these applications are data-oriented. They react continuously to their input dataflows and also produce some output dataflows. Dataflow programming is commonly used in these systems, and the main objective is to explore the parallelism of an application.

The dataflow model is an efficient model to represent signal processing appli-

cations. Such applications can be decomposed into a collection of operations that communicate each other, and they are easily represented by graphs. The dataflow model represents a program in a directed graph. Nodes of the graph represent operations to be performed (an instruction or a group of instructions), while data flow on edges of the graph and form the input to the nodes. The data are carried by tokens. Dataflow programming is sometimes coupled to functional programming languages in which the operations are evaluated as functions without side effect (no modification of state, no interaction with the outside world) and the output tokens are results of a function by consuming the input tokens.

There are many graph models based on the dataflow principle. Modeling program in this form does not specify the invocation rules of the operations and the techniques to model the communication channels. We present some important models as follows.

Kahn Process Networks (KPN)

Kahn described a simple programming language to model the parallel programming for distributed systems at the beginning of 70s [Kah74]. This model is known as Kahn Process Network (KPN). In such a process network, concurrent processes communicate by asynchronous message passing through unidirectional FIFO channels of infinite capacity. Each FIFO channel carries a sequence of tokens (possibly infinite) that evolve over time. A process is a mathematical function from a set of sequences to another set of sequences. Each token is written exactly once to a channel and is also read exactly once from a channel. Writing to a channel is non-blocking because of the channel's infinite capacity. However, reading from a channel is blocking, which means a process attempting to read from an empty channel stalls until this channel has sufficient tokens to be read.

A process in the KPN model is usually continuous. The continuity of process is a sufficient condition to ensure the determinacy of process networks. The network is determinate when the execution order of the processes in the network has no influence on the output result. It is shown in [KM77] that the continuity is guaranteed by the blocking reading of the FIFOs in practice.

While KPN was developed to model the concurrency in a program, the dataflow model in [Den74] was initially applied in the development of computer architectures. Operations of a graph are specified by actors in this dataflow model. An actor maps input tokens into output tokens when it fires. A set of firing rules specify when an actor can fire. A firing consumes input tokens and produces output tokens. A sequence of

firings is a particular type of Kahn process and is called a dataflow process [LP95]. A network of such processes is called a dataflow process network and is a particular case of KPN. Some special dataflow process models are presented as follows.

Synchronous Dataflow (SDF)

The Synchronous Dataflow (SDF) was firstly introduced in [LM87a, LM87b]. The SDF model is a special case of the dataflow process model. An actor is a function that fires when there are enough input tokens available to perform a computation (actors-lacking inputs can be invoked at any time). When an actor fires, it consumes a fixed number of new input tokens on each input edge. An actor is said to be synchronous if we can specify a priori the number of input tokens consumed on each input edge and the number of output tokens produced on each output edge each time the actor fires. An SDF actor is a dataflow actor that only contains a single firing rule [LP95]. This firing rule is valid for all possible numbers of tokens. The number of tokens consumed and produced is constant each time the actor fires. An SDF graph is a network of synchronous actors. Figure 2.1 shows an SDF actor and an SDF graph.

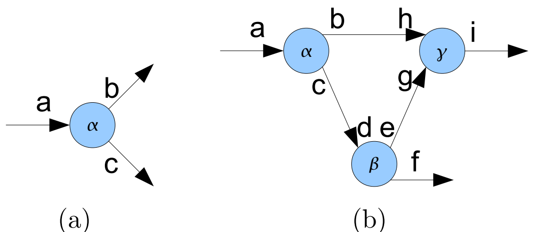


Figure 2.1: *(a) An SDF actor; (b) An SDF graph*

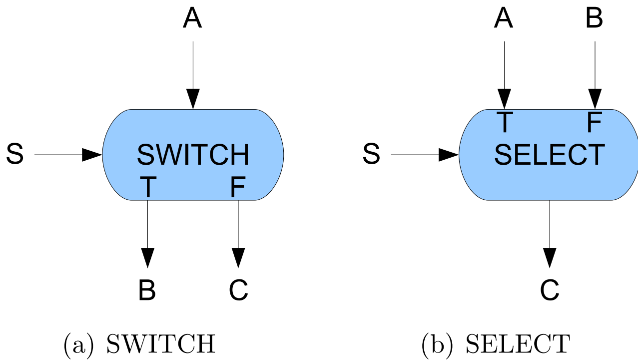
An SDF graph is well suited to model synchronous multirate signal processing applications. An application modeled by an SDF graph can be scheduled on single or multiple processors. It is known at compile time whether an SDF graph can be scheduled statically (at compile time) or not [LM87a]. When the schedule is determined, the memory usage is also bounded and known at compile time, then actors fire repeatedly during the execution.

Boolean Dataflow (BDF)

Although the SDF model is well suited to represent many parts of an application, it is usually difficult to represent an entire application by this model. For example,

control structures are very common in an application, and the execution of the application depends on the control input. The Boolean Dataflow (BDF) [Buc93] model is an extension of the SDF model, and it allows modeling a number of control structures by adding some specific control actors. These control actors have the dynamic comportments. The number of tokens consumed or produced can not be known a priori and depends on the value of an input control token.

SWITCH and SELECT are two control actors in the BDF model as shown in Figure 2.2. The SWITCH actor consumes an input token (A in Figure 2.2(a)) and a control token (S in Figure 2.2(a)). If the control token is TRUE, the input token is copied to the output labeled T; otherwise it is copied to the output labeled F. The SELECT actor performs the inverse operation, reading a token from the input labeled T if the control token is TRUE, otherwise reading from the input labeled F, and copying the token to the output.



(a) SWITCH (b) SELECT

Figure 2.2: *Control actors: SWITCH and SELECT*

The main advantage of the BDF model is that it allows modeling a larger class of programs. In fact, the BDF model is Turing-complete [Buc93], and any algorithm can be expressed in this model in principle. Unfortunately, it has been shown that the use of bounded memory and the presence of deadlocking are indeterminate at the compile time in general for the BDF model.

Dynamic Dataflow (DDF)

Many complex applications (e.g. multimedia) need to make decisions during the execution by concerning the treatment results. Therefore, some operations depend on the value of data (data-dependent). We have seen that the SDF model can be extended to BDF model by adding two dynamic actors of SWITCH and SELECT to

express control structures. Since the BDF model is Turing-complete, it is possible to model all functions that can be defined by an algorithm. Then it is possible to model all complex applications in the BDF model. However, this model is not convenient to express control structures other than structures like if-then-else. For example, a loop can be modeled but requires a really complex graph. The Dynamic Dataflow (DDF) model is then presented to facilitate the expression of control structures. This model may contain dynamic actors in addition to SWITCH and SELECT of the BDF model. They can consume and/or produce variable number of tokens according to their input tokens.

The DDF network is the most general dataflow process network. Thus the SDF and BDF models are special cases of the DDF model. Since a dataflow process network is a particular case of KPN, the containing relation of these four dataflow models is shown in Figure 2.3.

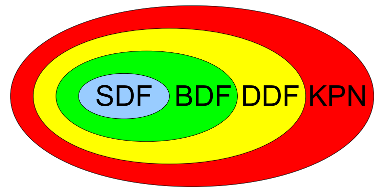


Figure 2.3: *Containing relation of different dataflow models*

The dataflow model is usually used to represent an application algorithm for programming. When this algorithm is implemented on a multi-processor system, scheduling is needed to assign operations on different processors of this system. In fact, we usually use another graph model to describe the algorithm for the scheduling problem. This model is called the DAG model and is explained in the next subsection.

2.2.2 DAG Model

DAG model is usually used to represent an algorithm for task scheduling. Therefore, a DAG is also called a task graph and is defined as follows.

DAG

A DAG is a directed acyclic graph $G = (V, E, w, c)$ where V is the set of nodes and E is the set of edges. A node represents a computation. For a pair of nodes $(n_i, n_j) \in V^2$, e_{ij} denotes the edge from the origin node n_i to the destination node n_j. e_{ij} represents the communication between the node n_i and the node n_j. The

weight of node n_i is denoted by $w(n_i)$ with $w(n_i) \in \mathbf{Q}^+$ ($\mathbf{Q}^+$ is the set of positive rational numbers) and represents the computation cost; the weight of edge e_{ij} is denoted by $c(e_{ij})$ with $c(e_{ij}) \in \mathbf{Q}_0^+$ ($\mathbf{Q}_0^+$ is the set of non-negative rational numbers) and represents the communication cost. A communication is not needed when the origin node and the destination node are assigned to the same processor, then the communication cost becomes null in this case.

In this model, the set $\{n_x \in V : e_{xi} \in E\}$ of all direct predecessors of node n_i is denoted by $pred(n_i)$; the set $\{n_x \in V : e_{ix} \in E\}$ of all direct successors of node n_i is denoted by $succ(n_i)$. A node n with $pred(n) = \phi$ is named a source node, and the set of all the source nodes of G is denoted by $source(G)$. A node n with $succ(n) = \phi$ is named a sink node, and the set of all the sink nodes of G is denoted by $sink(G)$.

Path

A path p in a DAG $G = (V, E, w, c)$ from a node n_1 to a node n_k is a sequence $\langle n_1, n_2, \ldots, n_k \rangle$ of nodes where the nodes are connected by edges $e_{i,i+1} \in E, i = 1, 2, \ldots, k-1$. This path is denoted by $p = p(n_1, n_k) = \langle n_1, n_2, \ldots, n_k \rangle$. A node n_i on the path p is denoted by $n_i \in p$; an edge e_{ij} on the path p is denoted by $e_{ij} \in p$.

A DAG can not contain any path from n_1 to n_k with $n_1 = n_k$, which is the meaning of "Acyclic". In fact, all the nodes of a path $p = \langle n_1, n_2, \ldots, n_k \rangle$ of a DAG must be different because a subsequence of p is also a path and therefore must be acyclic.

Topological Order

A topological order of a DAG $G = (V, E, w, c)$ is a linear ordering of all its nodes so that if it exists an edge $e_{ij} \in E$, then $n_i \in V$ must appear before $n_j \in V$ in the ordering.

A DAG can have multiple topological orders. The Depth First Search (DFS) algorithm given in [CLRS01] can be modified to sort nodes into a node list with topological order. This method is shown in Algorithm 2.1 and 2.2 with the complexity of $O(V + E)$. Some other topological orders will be given in the following sections and will be used for list scheduling heuristics in the following chapters.

Figure 2.4 gives a DAG example used in [KA99b] to illustrate the performances of different list scheduling methods. Figure 2.5 gives one topological order (from left to right) of this DAG. This DAG will also be used in the following chapters to show the performance of our methods.

Algorithm 2.1: **Topological_Sort**(G)

Input: A DAG $G = (V, E, w, c)$
Output: A node list with topological order

```
1 Create an empty node list NL;
2 for each n_i ∈ V do
3 |   Mark n_i as not discovered;
4 end
5 for each n_i ∈ V do
6 |   if n_i not discovered then
7 |   |   DFS_Visit(n_i,NL);
8 |   end
9 end
```

Algorithm 2.2: **DFS_Visit**(n_i,NL)

Input: A node $n_i \in V$ and a node list NL
Output: The modified node list NL

```
1 Mark n_i as discovered;
2 for each n_j ∈ succ(n_i) do
3 |   if n_j not discovered then
4 |   |   DFS_Visit(n_j,NL);
5 |   end
6 end
7 Insert n_i into the front of the node list NL;
```

2.2.3 DAG Properties

This section gives some properties of DAG that will be considered in the following chapters.

Path Length

Given a DAG $G = (V, E, w, c)$, the length of a path p in G is the sum of the weights of its nodes and edges:

$$len\left(p\right) = \sum_{n_i \in p, n_i \in V} w\left(n_i\right) + \sum_{e_{ij} \in p, e_{ij} \in E} c\left(e_{ij}\right)$$

The computation length of a path p in G is the sum of the weights of its nodes:

$$len_{comp}\left(p\right) = \sum_{n_i \in p, n_i \in V} w\left(n_i\right)$$

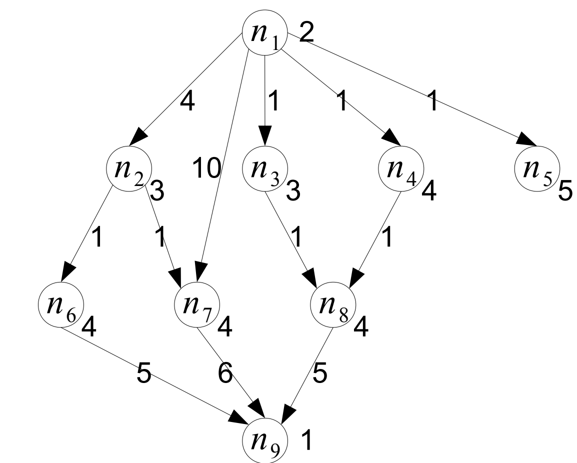


Figure 2.4: *A DAG example*

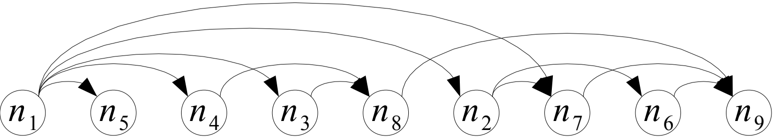


Figure 2.5: *A topological order of the DAG in Figure 2.4*

For the DAG given in Figure 2.4, if we choose $p = p(n_1, n_9) = \langle n_1, n_2, n_7, n_9 \rangle$, then $len(p) = 21$ and $len_{comp}(p) = 10$.

The path length $len(p)$ is the minimum execution time of a path p when all its communications exists, which could happen when each node is assigned to a different processor. Similarly, the computation path length $len_{comp}(p)$ is the minimum execution time of a path p when none of its communications exists. This case happens only when all the nodes of the path p are assigned to the same processor. The path length becomes the computation path length when all the communication costs are zeros.

Critical Path (CP)

Given a DAG $G = (V, E, w, c)$, a critical path cp of G is the longest path in G

$$len(cp) = \max_{p \in G} \{len(p)\}$$

A computation critical path cp_{comp} of G is the longest path in G

$$len_{comp}(cp) = \max_{p \in G} \{len_{comp}(p)\}$$

A critical path always starts at a source node and finishes at a sink node. In fact,

if a path does not start at a source node, we can add a predecessor of the first node to this path, and the path length is prolonged because the added node has a positive weight. Since a critical path has the maximum length, it must not be prolonged and therefore must start at a source node. Similarly, we can prove a critical path must finish at a sink node. The computation critical path must also start at a source node and finish at a sink node.

Node Levels

Given a DAG $G = (V, E, w, c)$, the top and bottom levels of $n_i \in V$ is defined as follows:

– **Top level**

The top level $tl(n_i)$ of n_i is the path length of the longest path ending at n_i, excluding $w(n_i)$

$$tl(n_i) = \max_{n_k \in source(G)} \{len(p(n_k, n_i))\} - w(n_i)$$

– **Bottom level**

The bottom level $bl(n_i)$ of n_i is the path length of the longest path starting at n_i

$$bl(n_i) = \max_{n_k \in sink(G)} \{len(p(n_i, n_k))\}$$

Similar to the critical path, the longest path ending at n_i must start at a source node, and the longest path starting at n_i must finish at a sink node. Figure 2.6 illustrates the top and bottom level of node n_i, where the longest path is dotted.

Using the computation path length can give another group of node levels, which are named computation top level and bottom level and are given as follows:

– **Computation top level**

The computation top level $tl_{comp}(n_i)$ of n_i is the computation path length of the longest path ending at n_i, excluding $w(n_i)$

$$tl_{comp}(n_i) = \max_{n_k \in source(G)} \{len_{comp}(p(n_k, n_i))\} - w(n_i)$$

– **Computation bottom level**

The computation bottom level $bl(n_i)$ of n_i is the computation path length of

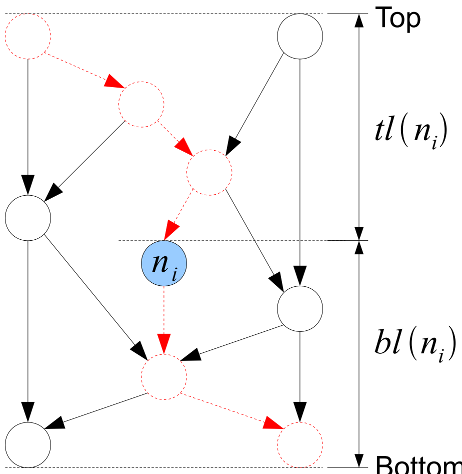


Figure 2.6: *Top and bottom levels*

the longest path starting at n_i

$$bl_{comp}(n_i) = \max_{n_k \in sink(G)} \{len_{comp}(p(n_i, n_k))\}$$

The two groups of top level and bottom level can also be defined recursively. Figure 2.7 illustrates the dependency between nodes to recursively define different top levels and bottom levels, where the dotted nodes and dotted edges are used to define the top levels and bottom levels of n_i.

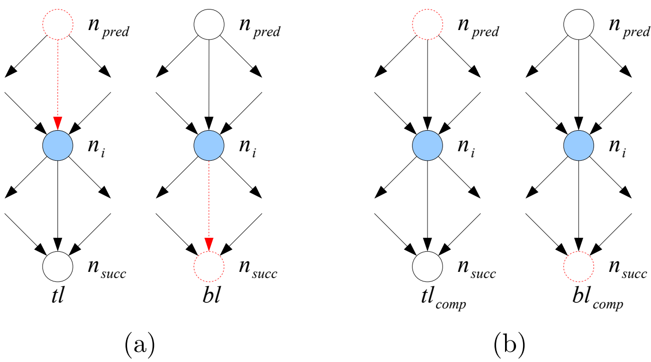


Figure 2.7: *Node dependency of the recursive definition of node levels*

– **Top and bottom levels** (Figure 2.7(a))

$$tl(n_i) = \begin{cases} 0, \text{if } n_i \in source(G) \\ \max\limits_{n_k \in pred(n_i)} \{tl(n_k) + w(n_k) + c(e_{ki})\}, \text{otherwise} \end{cases}$$

$$bl(n_i) = \begin{cases} w(n_i), \text{if } n_i \in sink(G) \\ \max\limits_{n_k \in succ(n_i)} \{bl(n_k) + c(e_{ik})\} + w(n_i), \text{otherwise} \end{cases}$$

– **Computation top and bottom levels** (Figure 2.7(b))

$$tl_{comp}(n_i) = \begin{cases} 0, \text{if } n_i \in source(G) \\ \max\limits_{n_k \in pred(n_i)} \{tl_{comp}(n_k) + w(n_k)\}, \text{otherwise} \end{cases}$$

$$bl_{comp}(n_i) = \begin{cases} w(n_i), \text{if } n_i \in sink(G) \\ \max\limits_{n_k \in succ(n_i)} \{bl_{comp}(n_k)\} + w(n_i), \text{otherwise} \end{cases}$$

Algorithm 2.3 and Algorithm 2.4 respectively compute the top levels and bottom levels by visiting each node in topological order and inverse topological roder.

Algorithm 2.3: **Compute_Top_Level**(G)

Input: A DAG $G = (V, E, w, c)$
Output: Top levels for each node of G

1 $NL \leftarrow$ **Sort_Topological**(G);
2 **for** *each* $n_i \in NL$ *from front to back* **do**
3     $max \leftarrow 0$;
4   **for** *each* $n_k \in pred(n_i)$ **do**
5         $max \leftarrow \max\{max, tl(n_k) + w(n_k) + c(e_{ki})\}$;
6   **end**
7     $tl(n_i) \leftarrow max$;
8 **end**

These two algorithms can also be used to compute the computation top level and computation bottom level by replacing the corresponding items for *max* according to the recursive definitions. The total repetition times in the for-loops of the two algorithms are the numbers of nodes and their predecessors/successors, which is in

Algorithm 2.4: **Compute_Bottom_Level**(G)

Input: A DAG $G = (V, E, w, c)$
Output: Bottom levels for each node of G

```
1 NL ← Sort_Topological(G);
2 for each n_i ∈ NL from back to front do
3     max ← 0;
4     for each n_k ∈ succ(n_i) do
5         max ← max {max, bl(n_k) + c(e_ik)};
6     end
7     bl(n_i) ← max + w(n_i);
8 end
```

total $O(V + E)$. Since sorting nodes in topological order with DFS has a complexity of $O(V + E)$, the total complexity of computing node levels is $O(V + E)$.

Relation between Critical Path and Node Levels

Since the bottom level is the path length of the longest path starting at a node, the maximum bottom level gives the path length of the longest path of a DAG, and therefore is the path length of the critical path. In fact, the sum of top level and bottom level of a node gives the path length of the longest path that passes this node in a DAG. Therefore, a node with the maximum sum of its top level and bottom level is on a critical path of the DAG. Similar relation can be obtained for the computation critical path and the computation top/bottom levels.

Tabel 2.1 summarizes different node levels of the DAG in Figure 2.4. The critical path length is 23, and n_1, n_7 and n_9 are nodes on a critical path. The computation critical path length is 11, and n_1, n_4, n_8 and n_9 are nodes on a computation critical path.

Table 2.1: *Different node levels for the DAG in Figure 2.4*

n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
tl	0	6	3	3	3	10	12	8	22
bl	23	15	14	15	5	10	11	10	1
$tl + bl$	23	21	17	18	8	20	23	18	23
tl_{comp}	0	2	2	2	2	5	5	6	10
bl_{comp}	11	8	8	9	5	5	5	5	1
$tl_{comp} + bl_{comp}$	11	10	10	11	7	10	10	11	11

Relation between Topological Order and Node Levels

The topological order is important for task scheduling heuristics, and the order of nodes usually affects the final result. Since a DAG can have multiple topological orders, it is necessary to get reasonable topological orders used for task scheduling heuristics. In fact, sorting nodes according to their levels can give useful topological orders.

Theorem 1. *Given a DAG $G = (V, E, w, c)$, the non-decreasing order of top level and the non-increasing order of bottom level are both topological orders; the non-decreasing order of computation top level and the non-increasing order of computation bottom level are also topological orders.*

Proof. In fact, a node always has a greater top level than its predecessors according to the recursive definition. Therefore, a node always appears after its predecessors in the non-decreasing order of top level, and this order satisfies the definition of topological order. Similarly, a node always has a greater bottom level than its successors according to the recursive definition. Therefore, a node always appears before its successors in the non-increasing order of bottom level, and this order also satisfies the definition of topological order.

Similar to the top level and bottom level, the computation top level and computation bottom level also give topological orders. □

The obtained orders based on the four node levels are usually different. Table 2.2 gives four node orders based on the four node levels for the DAG in Figure 2.4 according to the node levels in Table 2.1. Nodes with the same node level can be sorted randomly among themselves like $\{n_3, n_4, n_5\}$ for tl.

Table 2.2: *Different topological orders*

Node level	Node order
tl	$n_1, \{n_3, n_4, n_5\}, n_2, n_8, n_6, n_7, n_9$
bl	$n_1, \{n_2, n_4\}, n_3, n_7, \{n_6, n_8\}, n_5, n_9$
tl_{comp}	$n_1, \{n_2, n_3, n_4, n_5\}, \{n_6, n_7\}, n_8, n_9$
bl_{comp}	$n_1, n_4, \{n_2, n_3\}, \{n_5, n_6, n_7, n_8\}, n_9$

2.3 Architecture Model

On the other side, algorithm implementation on embedded systems with parallel architecture needs an architecture model. This section presents the classification of parallel architectures as well as some simple models. We also give an advanced architecture model to describe parallel embedded systems. This model is more accurate than the simple models and will be used in Chapter 3 for task scheduling.

2.3.1 Parallel Architectures

Flynn's taxonomy [Fly66] is the most popular classification of computer architecture. It defines four categories of computer architecture according to the concurrency of instruction and data streams. These categories are listed as follows:

- Single Instruction, Single Data stream (SISD)
- Single Instruction, Multiple Data stream (SIMD)
- Multiple Instruction, Single Data stream (MISD)
- Multiple Instruction, Multiple Data stream (MIMD)

Classical single-processor systems belong to the category of SISD. A system of SIMD treats multiple data streams by using a single instruction stream, and it usually describes a processor array. In a system of MISD, multiple instructions operate on a single data stream. This kind of architecture is rarely used in real systems. Modern parallel systems usually belong to the category of MIMD. MIMD is also divided into two groups: shared memory architecture and distributed memory architecture.

Shared Memory Architecture

In a shared memory architecture, multiple processors connect to a shared global memory via an interconnection network as shown in Figure 2.8. The interconnection network can be designed by using a bus or multiple buses, it can also be designed by using a switch. Communication between processors is achieved by writing/reading the shared memory. This kind of architecture is modeled as parallel random access machine (PRAM) in [FW78]. The PRAM model is simple for algorithm design and complexity analysis, but it is not accurate to describe real systems because of the competitions and delays for writing and reading.

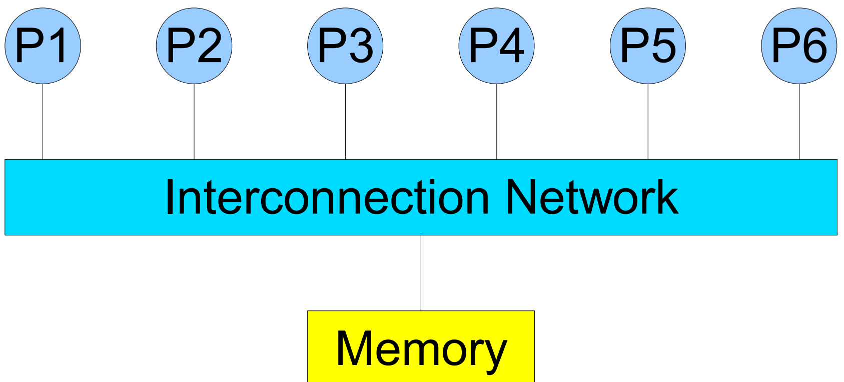


Figure 2.8: *Shared memory architecture*

Distributed Memory Architecture

A distributed memory architecture is also known as a message passing architecture. Figure 2.9 shows a distributed memory architecture where each processor has a local memory. A processor and its local memory are usually bound together as a unique processor module. A processor also includes a communication unit, and it communicates with another processor by sending/receiving messages.

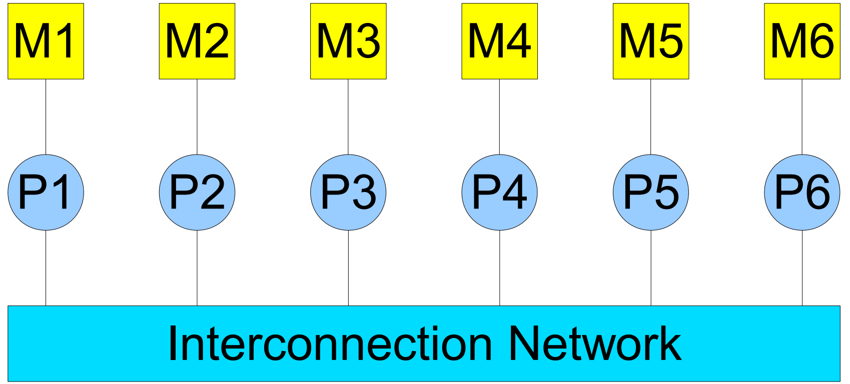
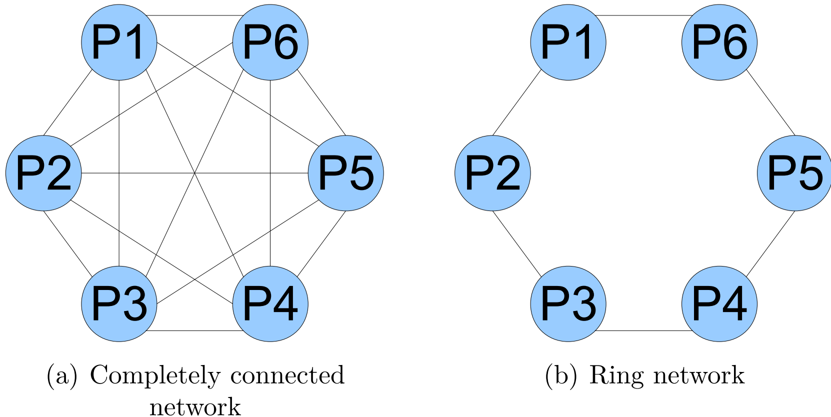


Figure 2.9: *Distributed memory architecture*

Distributed memory architectures are modeled differently according to the structure of the interconnection network. A distributed memory architecture can use static networks to interconnect processors via links. It can also use dynamic networks with switches. Figue 2.10(a) shows a completely connected static network for the distributed memory architecture given in Figure 2.9. The completely connected network is usually used to model a distributed memory architecture for performance analysis and estimations. Many works of task scheduling for parallel computation are based

on completely connected networks. However, the network is not really completely connected in all the cases (e.g. Figure 2.10(b)).



(a) Completely connected network

(b) Ring network

Figure 2.10: *Examples of static networks*

Parallel embedded systems usually use distributed memory architectures. Though the completely connected model is simple for performance analysis and estimations, it is not suitable to accurately describe parallel embedded systems because a parallel embedded system is usually different from a general parallel computer system. In fact, an embedded system usually has some specific properties and constraints. We need an appropriate model to describe the distributed memory architecture in order to reflect these properties and constraints. Therefore, we propose an advanced architecture model to describe parallel embedded systems in the following subsection.

2.3.2 Advanced Architecture Model

Rapid prototyping of parallel embedded systems needs abstract models to describe architectures and to facilitate the programming. An architecture should be modeled by considering the properties and constraints of embedded systems. It consists in modeling different components like processors, coprocessors, communication links as separate modules and describing their necessary properties. We model an architecture as a hypergraph, and components in the architecture becomes vertices and edges in the hypergraph. A parallel embedded system usually includes the following component models.

– **Processor**

A processor is a component that executes operations for computation and communication. Processors are one class of operators on which operations are exe-

cuted sequentially. A processor can only be used for one operation at a time. A processor contains an internal memory that can be read and written for communication. A processor can firstly configure a communicator to perform a communication from/to this processor; when the configuring is finished, the processor can execute another computation; thus the computation and the communication become parallel on this processor. A processor is a vertex in the hypergraph.

– **IP Coprocessor**
  Belonging to another class of operators, an IP coprocessor is a component usually designed with parallelism and pipeline. It is used for a specific operation that usually needs much more time if executed on a general processor. Therefore, this kind of operation is usually constrained on the IP coprocessor to shorten the execution time. An IP coprocessor can not configure a communicator; it needs another general processor to configure a communicator to perform the communication from/to it, and a processor can also perform the communication from/to the IP coprocessor without using a communicator. An IP coprocessor is a vertex in the hypergraph.
– **Memory**
  A memory is used to store data during the execution of a program. A memory is a slave terminal and can only be accessed by processors and communicators via communication nodes and links. A memory can have multiple ports with each port connecting to one communication link. The speed of reading/writing a memory depends on the bandwidth of the link. A memory is a vertex in the hypergraph.
– **Communicator**
  A communicator is a component only used to perform communications, and no computation can be executed on it. A communicator, which is usually a DMA controller, is configured by a processor before performing a communication. Communications on a communicator are done sequentially. A communicator is a vertex in the hypergraph.
– **Communication Node**
  A communication node is used to connect communication links, and no computation can be executed on it. A communication node is a vertex in the hypergraph. A communication node usually models a switch. The communication node is considered ideal and is described as follows:
  For a communication node cn_i, let $l_1, l_2, \ldots, l_n$ be all the communication links

connected to cn_i. If two links l_{i_1} and l_{i_2} of them are not used for the moment, a communication can be transferred on l_{i_1} and l_{i_2} without any impact from/to communications on other communication links connected to cn_i.

Communication nodes are contention-free according to this description: separate communication links connected to the same communication node can be used for different communications at the same time; however if a communication link is busy, a new communication can not begin on this link.

– **Bus**

A bus is a hyperedge in the hypergraph, which means multiple vertices can be connected together by this edge. Data can be flowed among all the vertices connected to it. However, the bus can only be used to transfer data between two vertices at a given time, and all other vertices must wait until the bus is free.

– **FIFO**

A FIFO is a directed edge from one vertex to another in the hypergraph. Data are flowed through a FIFO from the origin vertex to the destination vertex, and the opposite is impossible. A bidirectional FIFO should be presented as two FIFOs with opposite directions.

Figure 2.11 gives the legend of different vertices and edges. These components are used in the advanced architecture model.

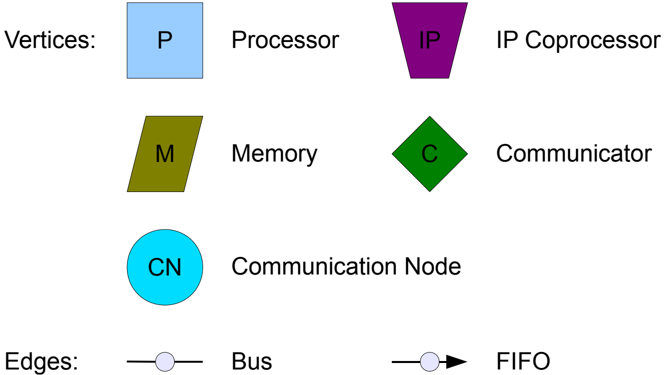


Figure 2.11: *Legend of vertices and edges*

Definition of Advanced Architecture Model

As shown in Figure 2.12, we model an architecture as a hypergraph that is denoted by

$$\begin{aligned} Archi &= (P, IP, M, C, CN, B, F, PF) \\ &= (T, CP, CN, L, PF) \end{aligned}$$

where P, IP, M, C and CN are five sets of different vertices; B and F are two sets of different edges; PF is a set of property functions for the architecture and contains four functions of c, s, a and b.

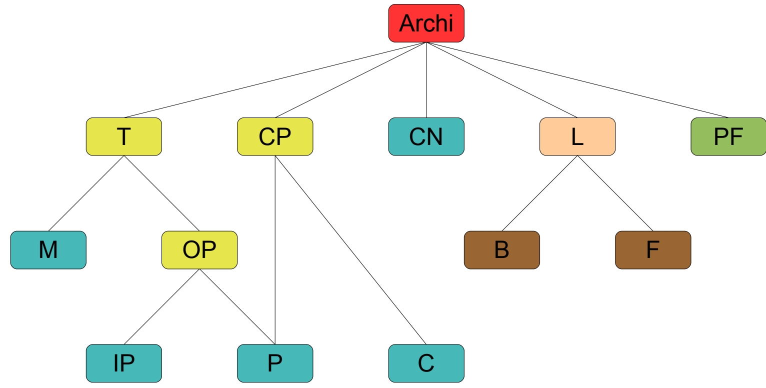


Figure 2.12: *Organization of the architecture model*

The union of P and IP is designated the operator set OP, $OP = P \cup IP$; the union of OP and M is designated the terminal set T, $T = OP \cup M$; the union of C and P is designated the communication performer set CP, $CP = C \cup P$; and the union of B and F is designated the communication link set L, $L = B \cup F$.

The elements in the sets of vertices and edges are explained as follows:

– A vertex of $p_i \in P$ represents a processor;
– A vertex of $ip_i \in IP$ represents an IP coprocessor;
– A vertex of $m_i \in M$ represents a piece of memory;
– A vertex of $c_i \in C$ represents a communicator;
– A vertex of $cn_i \in CN$ represents a communication node;
– An edge of $b_i \in B$ represents a bus;
– An edge of $f_i \in F$ represents a FIFO.

c, s, a and b are four functions used for describing properties of a parallel embedded system. They are described in detail as follows:

– The function $c : P \mapsto \{C_i | C_i \subseteq C\}$ gives the configurability of a processor. $c(p_i)$ is the set of communicators that can be configured by p_i.

- The function $s : P \times C \mapsto \mathbf{Q}^+$ gives the setup time for a processor to configure the communicator before the communication being performed, where $\mathbf{Q}^+$ is the set of positive rational numbers. $s(p_i, c_j)$ is the time used by p_i to configure c_j with $c_j \in c(p_i)$.
- The function $a : T \mapsto \{CP_i | CP_i \subseteq CP\}$ gives the accessibility of a terminal by communication performers. $a(t_i)$ is the set of communication performers that can access t_i. When a processor p_i is used as communication performer, it can always access its internal memory, therefore $p_i \in a(p_i)$.
- The function $b : L \mapsto \mathbf{Q}^+$ gives the average data rate of a communication link. $b(l_i)$ is the number of bytes transferred per time unit.

Architecture Examples

The architecture model is used to model multiprocessor embedded systems. Figure 2.13 shows the functional block diagram of the C6474 high-performance multicore DSP of Texas Instruments in [Tex08]. The C6474 DSP mainly consists of

- three C64x+ DSP cores,
- two high-performance embedded coprocessors [enhanced Viterbi Decoder Coprocessor (VCP2) and enhanced turbo decoder coprocessor (TCP2)],
- one Switched Central Resource (SCR) including a 64-channel enhanced direct memory access (EDMA3.0),
- other peripherals.

These components are interconnected by buses of TMS320C6474 Common Bus Architecture, which have different data rates.

Figure 2.14 gives the architecture model for the C6474 DSP:

- Three processors (P1, P2, P3) model three C64x+ DSP cores;
- Two IP coprocessors (IP1, IP2) model the VCP2 and TCP2 coprocessors;
- One communication node (CN1) models the SCR; one communicator (C1) models the EDMA3.0;
- Other peripherals are omitted.

Any one of the three processors can configure the communicator, and all the five operators can be accessed by the communicator.

The C6474 Evaluation Module (EVM) [(1)] includes two C6474 multicore DSPs connected via high speed SERDES interfaces: Serial RapidIO (SRIO), Gigabit Ethernet MAC (GEMAC), and Antenna Interface (AIF). Figure 2.15 gives an architecture

(1). http://focus.ti.com/docs/toolsw/folders/print/tmdxevm6474.html

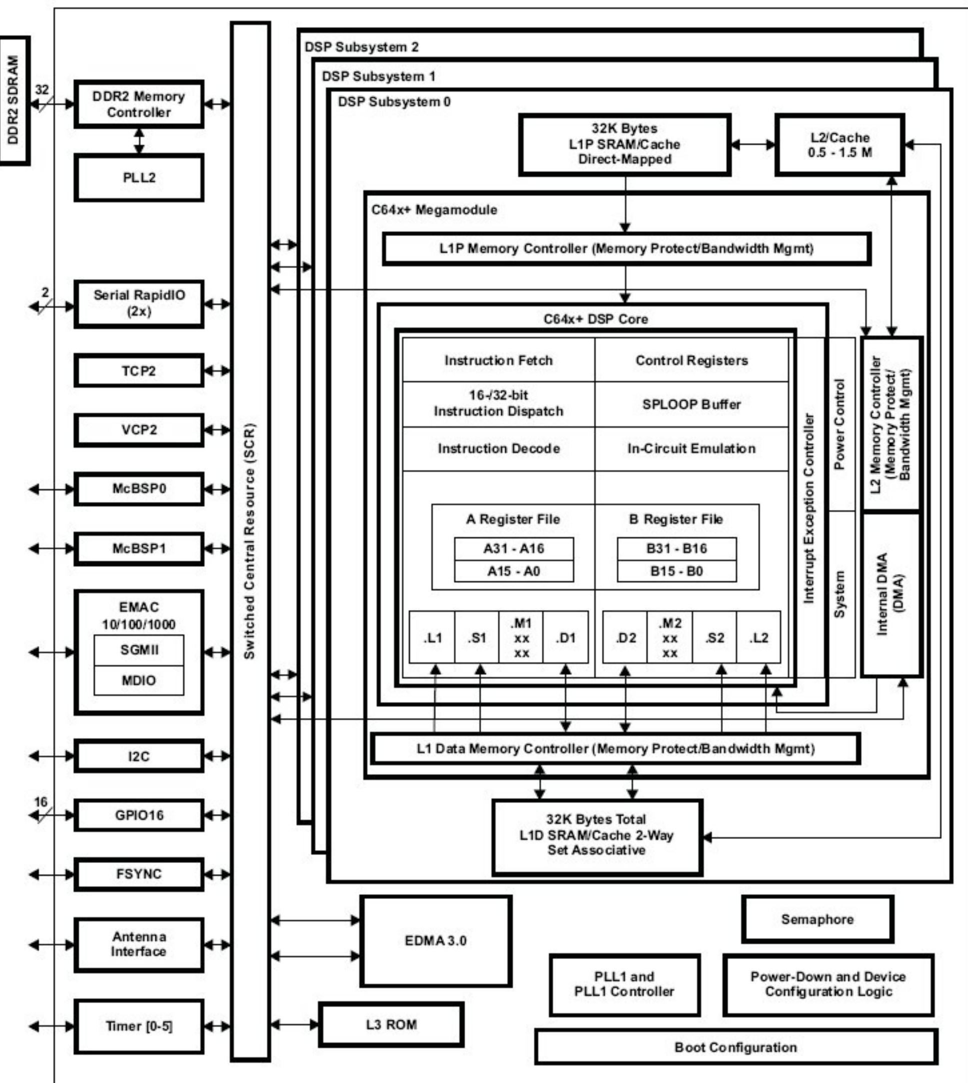


Figure 2.13: *Multicore DSP architecture of Texas Instruments*

model for this multi-DSP platform. The architecture is composed of two identical sub-systems that are interconnected by two FIFOs of SRIO. Each sub-system consists of a memory of DDR2 SDRAM and a C6474 DSP that has been modeled in Figure 2.14. The SRIO port of a C6474 DSP is indeed a bridge between the bus and the FIFOs, and it is modeled as a communication node. The three processors share the communicator in the sub-system, but they can not configure the communicator in the other sub-system. However, all the processors and memories can be accessed by the two communicators.

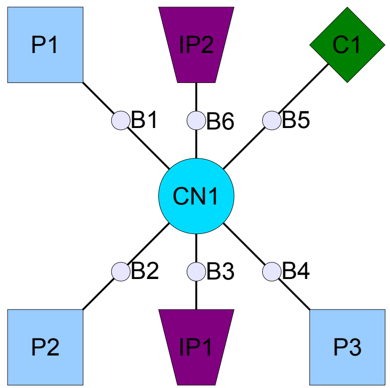


Figure 2.14: *Multicore DSP architecture of Texas Instruments*

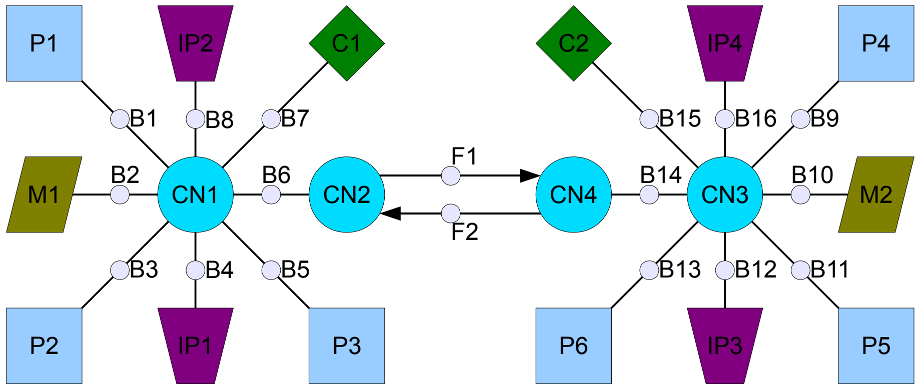


Figure 2.15: *A multi-DSP architecture*

The architecture model is also used to describe FPGA based MPSoC systems. Figure 2.16 gives such an example on the FPGA of Xilinx (2). Two MicroBlaze processors are interconnected by a bridge, and each processor has a DMA controller as communicator. An IP coprocessor is connected to P1 by two FIFOs of FSL (Fast Simplex Link), and another IP coprocessor is connected to P2 via a bus of PLB (Processor Local Bus).

2.3.3 Architecture Specification with IP-XACT Standard

Our advanced architecture model is used in PREESM and aims to describe the behavior of the most common components for embedded systems. It is consistent with the IP-XACT [SPI08] standard and can be edited in Graphiti.

The IP-XACT standard is specified by the Spirit Consortium (3) in order to de-

(2). http://www.xilinx.com/
(3). http://www.spiritconsortium.org

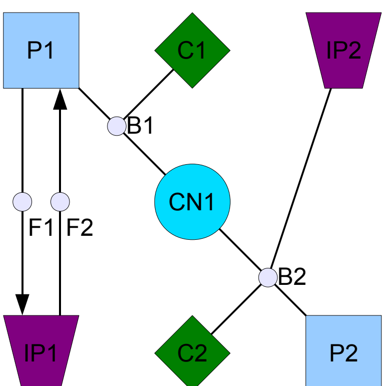


Figure 2.16: *MPSoC architecture on the FPGA of Xilinx*

scribe digital hardware architectures. The IP-XACT 1.4 release issued in 2006 defines a complex XML structure that aims at embracing the hardware description issues at both abstraction levels of Register Transfer Level (RTL) and Transaction-Level Modeling (TLM). In order to feed a static task scheduling process, we only use a subset of IP-XACT 1.4 focused on its high-level TLM capabilities. Our subset can describe TLM level architectures with several parameters. Configuration files in the Graphiti editor enable the edition of this subset of IP-XACT 1.4 as graphs.

An architecture is a <spirit:design> in an IP-XACT description specified in an XML file. A <spirit:design> usually consists of several component instances specified as <spirit:componentInstance>. Parameters are the most important information for an component instance and are specified in <spirit:configurableElementValue>. For example, a type parameter is associated to each component instance and is associated to a type of component in the advanced architecture model. Figure 2.17 gives an example of the IP-XACT description for a FIFO instance.

Component instances are connected by their interfaces that are edited as ports of vertices in Graphiti. An interconnection between two interfaces is specified by a <spirit:interconnection> in a <spirit:design>. A <spirit:interconnection> is usually used to connect a component to a bus and is undirected. Since the interconnection between a component and a FIFO is directed, we specify this kind of interconnections by adding a <spirit:displayName> of directed in <spirit:interconnection>. The accessing and configuring properties of the architecture model are also specified by <spirit:interconnection> but with different <spirit:displayName> of access and configure. Interconnections for accessing and configuring are also directed. The direction is from the first <spirit:activeInterface> to the second <spirit:activeInterface> for all the three directed interconnection. The code given in Figure 2.18 shows the code of an

```
<spirit:componentInstance>
   <spirit:instanceName>fifo1</spirit:instanceName>
   <spirit:componentRef spirit:library="preesm" spirit:name="Fifo"
    spirit:vendor="ietr" spirit:version="1.0"/>
   <spirit:configurableElementValues>
      <spirit:configurableElementValue spirit:referenceId="componentType">
       fifo</spirit:configurableElementValue>
      <spirit:configurableElementValue spirit:referenceId="refinement"/>
      <spirit:configurableElementValue spirit:referenceId="dataRate">
       1.0</spirit:configurableElementValue>
   </spirit:configurableElementValues>
</spirit:componentInstance>
```

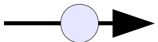


Figure 2.17: *Component instance of a FIFO*

interconnection from a processor (proc1) to a FIFO (fifo1).

```
<spirit:interconnection>
   <spirit:name/>
   <spirit:displayName>directed</spirit:displayName>
   <spirit:activeInterface spirit:busRef="o1" spirit:componentRef="proc1"/>
   <spirit:activeInterface spirit:busRef="i1" spirit:componentRef="fifo1"/>
</spirit:interconnection>
```

Figure 2.18: *Interconnection from a processor to a FIFO*

We give the codes for the specifation of the architectures in Figure 2.14 and Figure 2.16 in Appendix A.

2.4 Conclusion

This chapter introduced graph models for parallel embedded systems. Here algorithms and architectures are both modeled as graphs to facilitate the parallel programming.

An algorithm can be modeled as different graphs. We made a choice among these algorithm models, and the DAG model was chosen for task scheduling. In a DAG, nodes represent computations and edges represent communications between computations. We presented the properties of the DAG in detail.

Parallel architectures are usually classified as the shared memory architecture and the distributed memory architecture. We presented several graph models for these two types of architectures. However, these models are usually not accurate for parallel

embedded systems. Since the distributed memory architecture is usually used for parallel embedded systems, we proposed an advanced architecture model to describe heterogeneous parallel embedded systems. We use five kinds of vertices (processor, IP coprocessor, memory, communicator, and communication node) and two kinds of edges (bus and FIFO) to model different components of an embedded system. In addition, four functions are used to describe properties of the components.

The advanced architecture model can describe many real systems such as multicore DSP and FPGA based MPSoC. Architectures of these systems are specified with Graphiti by respecting the IP-XACT standard. The advanced architecture model will be used to describe the task scheduling problem in the next chapter.

3

Task Scheduling in Parallel Embedded Systems

3.1 Introduction

Parallelism is a solution to satisfy the requirement of great computation ability in embedded systems for modern digital signal processing and image processing applications. The work of distributing and scheduling tasks of a program over a parallel embedded system is not straightforward. When performed manually, it is usually time-consuming and the result is usually a suboptimal solution. Therefore, it is necessary to research automatic task scheduling methodologies that may produce near optimal results. The time consumed for the task scheduling should also be short.

Scheduling has been used in many domains [BK06, Bru07, BEP+07] and is specially discussed for parallel and distributed computing [Sar89, SB00, CDKM02, Sin07]. Task scheduling in parallel systems consists in assigning and ordering computations and communications respectively to processors and communication links of the target system in order to finish all the tasks as soon as possible. The scheduling can be static, which is done at compile time, or dynamic, which is done at run time. The static scheduling is more suitable than the dynamic one for signal processing applications like digital communication and video compression by leading to lower code size and higher computation efficiency. Since the static scheduling is usually used in the context of embedded systems, we only concern static scheduling in this work, and

all the task scheduling problems discussed in the following parts belong to the static scheduling.

This chapter is organized as follows: We first introduce the general task scheduling problem in parallel and distributed computing in Section 3.2, then the task scheduling with the architecture model is discussed in detail in Section 3.3. Since it is difficult to start with the advanced architecture model, we simplify the task scheduling with a topology graph model for architecture in Section 3.4. The simplified task scheduling is the task scheduling with communication contention, and we will research advanced heuristic techniques based on it in the following chapters. Section 3.5 gives the conclusion of this chapter.

3.2 General Task Scheduling

Task scheduling is an important aspect of parallel programming because the schedule result directly affects the parallel computation performance. Task scheduling consists in assigning computations and communications to components and finding time intervals on these components to execute the computations and communications. The aim of task scheduling is to get the shortest execution time which is also called the schedule length.

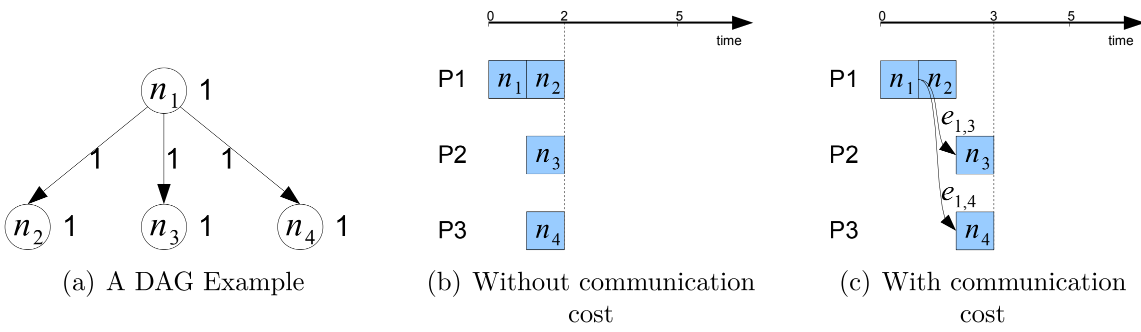
General task scheduling uses a DAG to model an algorithm where nodes represent computations and edges represent communications. An architecture includes multiple processors interconnected by a communication network. The execution of computations on a processor is sequential, and a computation can not be divided into several parts. A computation can not be started until all its input communications are finished, and all its output communications can not be started until it is finished.

3.2.1 Without/With Communication Costs

Communications are treated differently under different circumstances. Communication costs were not taken into account in 1970s [ACD74]. Then it was noticed that scheduling without communication costs was usually not accurate, and people started to consider communication costs in task scheduling [RS87, HCAL89, YG93, ERA94]. Though scheduling without communication costs is a special case of scheduling with communication costs, we will present them from special to general as the order of their appearance in the history.

Scheduling DAGs without Communication Costs

Since communication costs were not taken into account in early task scheduling problems, edges in a DAG only represent precedence constraints. When a computation is finished, its output communications can be immediately used in other computations even though these computations are executed on different processors. This circumstance can be considered to occur in an ideal shared memory architecture. In fact, since all processors use the shared memory, data are written to this memory when a computation is being executed. Once the computation is finished, another computation can read the data directly without additional time cost. Figure 3.1(a) gives a DAG example, and Figure 3.1(b) gives the scheduling on a 3-processor system without communication costs.



(a) A DAG Example (b) Without communication cost (c) With communication cost

Figure 3.1: *Scheduling without/with communication cost*

Since a real shared memory architecture is far away from an ideal model, task scheduling without communication cost is not accurate for parallel embedded systems. In fact, a parallel embedded system usually uses the distributed memory architecture and has to consider communication costs in task scheduling.

Scheduling DAGs with Communication Costs

Communication costs are considered for accurate task scheduling in distributed memory architecture. A general architecture model has the following properties:

- A (local) communication does not need time cost when its origin computation and destination computation are assigned to the same processor.
- A (remote) communication needs time cost when its origin computation and destination computation are assigned to different processors.
- Communications and computations are parallel. A processor is not involved for performing a communication.

– Processors are completely connected. Each pair of processors can communicate simultaneously without contention for communication resources.
– Multiple communications can be simultaneously performed between two processors.

Figure 3.1(c) gives the scheduling with communication costs for the DAG in Figure 3.1(a). Compared to the scheduling without communication costs in Figure 3.1(b), the schedule length increases from 2 to 3.

3.2.2 Scheduling Methodologies

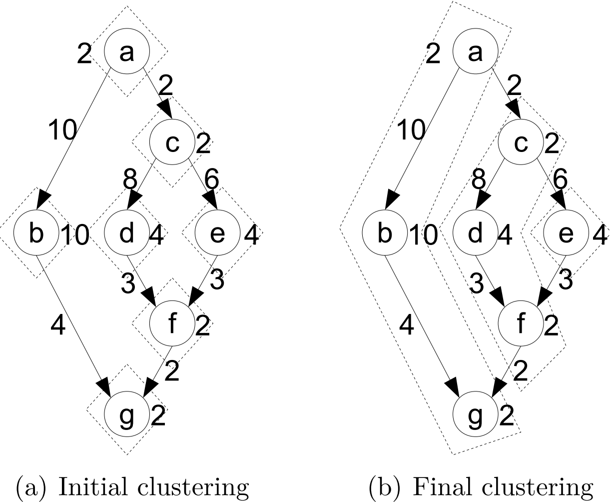
The general task scheduling problem has been proven to be NP-hard [GJ79, Sar89, Bru07], therefore, many works try to find heuristics to go up to the optimal solution. There are mainly two different scheduling methodologies: Clustering and List Scheduling.

Clustering

Clustering [GY93] is a kind of scheduling methodology on a virtual system with unlimited number of processors. The motivation of clustering was given in [Sar89]: if it is best to execute some computations on the same processors of an ideal system (a system with as many as possible processors), these computations should also be executed on the same processor in a real system. Linear clustering is a special class of clustering where only dependent nodes are grouped into one cluster. Figure 3.2 shows an example of linear clustering: Each node possess a cluster at the beginning (Figure 3.2(a)), and three clusters are obtained at last (Figure 3.2(b)).

Many heuristics have been reported for DAG clustering like the Edge-zeroing (EZ) algorithm [Sar89], the Dominant Sequence Clustering (DSC) algorithm [YG94], the Mobility Directed (MD) algorithm [WG90] and the Dynamic Critical Path (DCP) algorithm [KA96]. In [HM95], the clustering problem is treated as an integer linear program problem. The grain packing problem is essentially the clustering problem [ERLA94, KL88, MG89]. Clustering algorithms can be also used for scheduling on systems with limited number of processors like in [SL93b].

Though clustering is usually feasible in homogeneous systems, it is also proposed for heterogeneous systems [CJ01]. Clustering is often proposed as the first step to schedule for a limited number of processors. The second and third steps are respectively assigning and scheduling clusters on processors. In fact, a heterogeneous



(a) Initial clustering (b) Final clustering

Figure 3.2: *Linear clustering*

embedded system may include a processor that is designed for a special computation. If such a computation is grouped into a cluster with other computations, this cluster will be invalid because the other computations can not be executed on the same processor. Therefore, clustering is not very suitable for task scheduling in parallel embedded systems.

List Scheduling

List scheduling is a kind of scheduling methodology with limited number of processors. List scheduling can be classified as static or dynamic according to whether the node list is generated before or during the scheduling. Here, the "static" means that the node list is generated statically, and "dynamic" means that the node list is generated dynamically. The static list scheduling and dynamic list scheduling are both static scheduling because they are done at compile time.

Algorithm 3.1 shows the general static list scheduling method. It consists of two steps: (1) sorting nodes into a list in topological order, (2) schedule each node of the list onto a processor of the system.

Algorithm 3.2 gives the general dynamic list scheduling method. Though there is not a static node list, the order in which nodes are scheduled should also be a topological order.

Many list scheduling heuristics have been proposed since the early era of the scheduling problem such as the Highest Level First (HLF) algorithm [Hu61], the

Algorithm 3.1: **General_Static_List_Scheduling**(G, P)

Input: A DAG $G = (V, E, w, c)$ and a set of processors P
Output: A schedule of G on P

1 Sort nodes $n \in V$ into a list NL in topological order;
2 **for** *each* $n \in NL$ **do**
3   Select a processor $p \in P$ for n;
4   Schedule n on p;
5 **end**

Algorithm 3.2: **General_Dynamic_List_Scheduling**(G, P)

Input: A DAG $G = (V, E, w, c)$ and a set of processors P
Output: A schedule of G on P

1 $UnscheduledNodes \leftarrow V$;
2 **while** $UnscheduledNodes \neq null$ **do**
3   Choose a node $n \in UnscheduledNodes$;
4   Select a processor $p \in P$ for n;
5   Schedule n on p;
6   Remove n from $UnscheduledNodes$;
7 **end**

Highest Level First with Estimated Times (HLFET) algorithm [ACD74], the Critical Path/Most Immediate Successors First (CP/MISF) algorithm [KN84], the Modified Critical Path (MCP) algorithm [WG90], the Earliest Time First (ETF) algorithm [HCAL89] and the Dynamic Level Scheduling (DLS) algorithm [SL93a]. This kind of scheduling also appears in [GG69, YG93]. Though the architecture is a completely connected model, list scheduling can be also used in heterogeneous architecture with arbitrary interconnection network like in [SL93a, KA99a].

3.2.3 Advanced Techniques

Since task scheduling is very important for parallel computing, some advanced techniques are proposed in addition to the basic methodologies.

Node Duplication

Node duplication is an advanced scheduling technique and aims to shorten schedule length by reducing interprocessor communications. Figure 3.3 shows the scheduling for the DAG in Figure 3.1(a) with the node n_1 being duplicated on each processor. Since the node n_1 is duplicated on each processor, communications from n_1 to other

nodes are no longer necessary because these communications are all local communications. Therefore, the schedule length becomes 2.

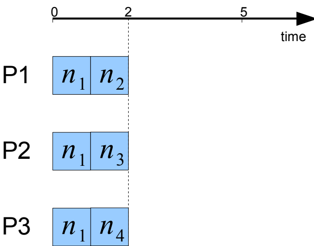


Figure 3.3: *Scheduling with node duplication*

Node duplication has been used in many scheduling approaches. Clustering heuristics with node duplication are proposed in [PY90, CC91, PLW96, LP98]. List scheduling heuristics with node duplication include the Duplication Scheduling Heuristic (DSH) algorithm [KL88], the Bottom-Up Top-Down Duplication heuristic (BTDH) algorithm [CR92] and the Critical Path Fast Duplication (CPFD) algorithm [AkK98]. Another list scheduling heuristic with node duplication is proposed in [HJ05] and can be used in heterogeneous systems.

Similar to the clustering, the node duplication technique is not practical in heterogeneous embedded systems where a processor is designed only for a kind of node. In fact, the general node duplication technique will become very difficult because a duplicated node can not be executed on such a special processor. Therefore, the node duplication technique is not very suitable for task scheduling in parallel embedded systems.

Search-based Methods

Since heuristics usually can not give the optimal result for task scheduling which is a NP-hard problem, search-based methods are used to approach the optimal result like the Fast Assignment using Search Technique (FAST) algorithm [KAG96] which uses a local search [Gu93, WM89, SG91] technique. Genetic algorithms [HH04, SD08] are another kind of random search methods, and they are used in [WSRM97, WYKH97, KA97, DAYA02, WYJ+04] for task scheduling. Search-based methods sacrifice time to get better results, and they are a complement to heuristics.

With Arbitrary Architectures

The interconnection network in a parallel embedded system is usually not completely connected. Therefore, the classic task scheduling methods shown above becomes inaccurate for scheduling on an arbitrary architecture, and communication contention needs to be considered on communication links. Communications are performed sequentially on a communication link, but different computations and communications may be executed simultaneously respecting the inputs and outputs constraints. Figure 3.3 shows the scheduling for the DAG in Figure 3.1(a) on the architecture in Figure 3.4(a). Communication contention occurs in the shared bus, and the schedule length becomes 4.

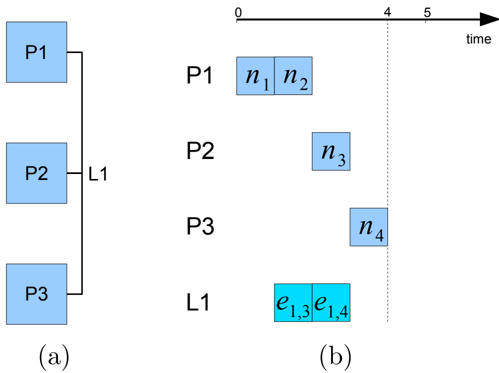


Figure 3.4: *Scheduling on an architecture with shared bus*

Some scheduling methods for arbitrary architecture have been proposed like the Mapping Heuristic (MH) algorithm [ERL90], the Dynamic Level Scheduling (DLS) algorithm [SL93a], the Bottom-Up (BU) algorithm [MG94] and the Bubble Scheduling and Allocation (BSA) algorithm [KA95]. An extension of the DLS algorithm is reported in [SSRM94], and another method called Latest Starting Time (LST) [KS93] is proposed for hypercube architecture. An arbitrary parallel system can be heterogeneous. Therefore, task scheduling in heterogeneous systems are also considered in some works like [MSP+95, OH96, KA99a, THW99, THW02, BBR02, SS04, LPX05]. Since we have given an advanced architecture model to accurately describe parallel embedded systems in Chapter 2, we need to reformulate the task scheduling problem with this new model and research heuristics based on it.

3.3 Task Scheduling with Advanced Architecture Model

We explore the task scheduling with our advanced architecture model for parallel embedded systems. As to the algorithm, the DAG model is used, where nodes and edges respectively describe computations and communications. Task scheduling with the advanced architecture model consists in assigning computations and communications to components and finding time intervals on these components to execute the computations and communications. A computation is executed on a processor or an IP coprocessor, and a communication is performed either by a communicator or a processor. If the communication is performed by a communicator, this communicator should firstly be configured by a processor, and then the data are transferred from the origin terminal to the destination terminal by the communicator through communication links and nodes. If the communication is performed by a processor, the processor can perform the communication immediately without a setup time, but the processor can not be used to execute a computation at the same time.

3.3.1 Routing with Architecture Model

Since the architecture is not completely connected (some vertices are not connected directly by edges), routing is necessary to transfer data from one terminal to another. We divide a route into several steps, and data are transferred from the origin terminal to the destination terminal step by step.

A route step contains a processor, a beginning terminal, a beginning communication link, a chain of communication nodes and links, and an ending terminal. The processor is used to initiate the communication on the route step, and it must be same to either the beginning or the ending terminal. If a communicator is used to perform the communication on this route step, the communicator is configured by the processor and is able to access both the beginning and ending terminals. If no communicator is used, the processor is used to perform the communication on this route step and must be able to access both the beginning and ending terminals. We represent a route step from the beginning terminal t_b to the ending terminal t_e as follows:

$$Rs\left(t_b, t_e\right) = \left\{processor, communicator_{0/1}, \langle t_b, l_0, \left(cn_k, l_k\right)_{0\rightarrow\infty}, t_e\rangle\right\}$$

where

- $communicator_{0/1}$ means 0 or 1 communicator is needed for this route step;
- l_0 is the first link from t_b;
- $(cn_k, l_k)_{0\to\infty}$ means 0 or a finite number of (cn_k, l_k) two-tuples are used to compose a chain from l_0 to t_e.

We define a route from the origin terminal t_{origin} to the destination terminal $t_{destination}$ as a list of route steps that is represented as follows:

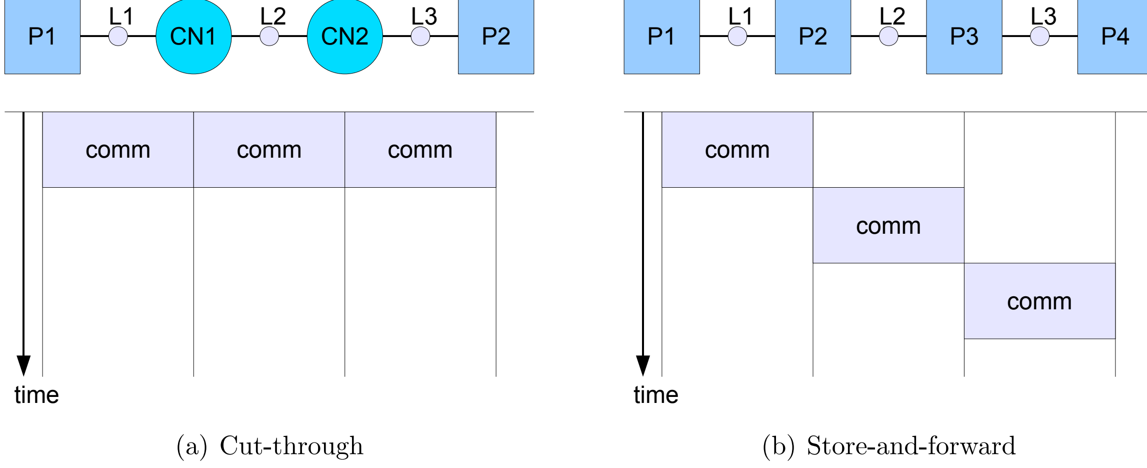
$$R\left(t_{origin}, t_{destination}\right) = \left\langle Rs\left(t_{b_1}, t_{e_1}\right), \ldots, Rs\left(t_{b_l}, t_{e_l}\right), \ldots, Rs\left(t_{b_m}, t_{e_m}\right)\right\rangle$$

This route consists of m steps to transfer the data. The route steps are constrainted by $t_{origin} = t_{b_1}$, $t_{e_1} = t_{b_2}, \ldots, t_{e_{m-1}} = t_{b_m}$, $t_{e_m} = t_{destination}$.

Since circuit switching is usually used for communications in embedded systems, which is different from the packet-based communication, a communication assigned on a route step must be aligned on all the communication links of this route step even if these links have different data rates. In addition, the communication can not be started on the next route step until it is finished on the current route step. Therefore, the communication is handled

- in the mode of cut-through on a route step;
- in the mode of store-and-forward between route steps.

These two modes have been used in computer networks [HP02] and are shown in Figure 3.5.



(a) Cut-through (b) Store-and-forward

Figure 3.5: *Two routing modes*

3.3.2 Scheduling with Advanced Architecture Model

The following terms describe a schedule S of a DAG $G = (V, E, w, c)$ on an architecture $Archi = (T, CP, CN, L, PF)$ (cf. Chapter 2).

The start time of a node $n_i \in V$ on an operator $op \in OP$ is denoted by $t_s(n_i, op)$; the finish time is given by $t_f(n_i, op) = t_s(n_i, op) + w(n_i, op)$, where $w(n_i, op)$ is the execution duration of n_i on op. Since execution durations of a node on different operators can be very different ($w(n_i, op_j) \gg w(n_i, op_k)$), this node is usually constrained to some operators which give relatively small execution durations. The set of operators on which n_i is constrained to be executed is denoted by $Oper(n_i)$, and the operator on which n_i is actually assigned is denoted by $oper(n_i)$. The node weight is given by $w(n_i) = \frac{1}{M}\sum\limits_{op} w(n_i, op)$, where M is the number of operator types in $Oper(n_i)$, and op represents a type of operator in $Oper(n_i)$.

The finish time of an operator is the maximum finish time among all nodes assigned on this operator, $t_f(op) = \max\limits_{oper(n_i)=op} \{t_f(n_i, oper(n_i))\}$, and the schedule length of S is the maximum finish time among all the operators in the system, $sl(S) = \max\limits_{op \in OP} \{t_f(op)\}$.

A communication represented by an edge $e_{ij} \in E$ of a DAG is needed only when its origin node n_i and its destination node n_j are assigned to different operators. Since a communication is transferred on a route consisting of several steps, the route used for an edge e_{ij} is denoted by $Route(e_{ij}) = \langle Rs(t_{b_1}, t_{e_1}), \ldots, Rs(t_{b_l}, t_{e_l}), \ldots, Rs(t_{b_m}, t_{e_m})\rangle$ with $t_{b_1} = oper(n_i)$ and $t_{e_m} = oper(n_j)$. A communication is differently treated on a route step according to its performer.

Performing Communication by Communicator

If a communicator c_l is used to perform a communication e_{ij} on a route step $Rs(t_{b_l}, t_{e_l})$, it must be configured by a processor p_l. The start time of e_{ij} on processor p_l of $Rs(t_{b_l}, t_{e_l})$ is denoted by $t_s(e_{ij}, p_l, Rs(t_{b_l}, t_{e_l}))$, and the finish time of e_{ij} on p_l is given by

$$t_f(e_{ij}, p_l, Rs(t_{b_l}, t_{e_l})) = t_s(e_{ij}, p_l, Rs(t_{b_l}, t_{e_l})) + s(p_l, c_l)$$

where $s(p_l, c_l)$ is the setup time for p_l to configure c_l.

The start time of e_{ij} on link l_k of $Rs(t_{b_l}, t_{e_l})$ is denoted by $t_s(e_{ij}, l_k, Rs(t_{b_l}, t_{e_l}))$, and the finish time of e_{ij} on link l_k is denoted by $t_f(e_{ij}, l_k, Rs(t_{b_l}, t_{e_l}))$. Since the communication is handled in the cut-through mode on a route step, e_{ij} is aligned on

all the links of this route step, that is

$$t_s\left(e_{ij}, l_0, Rs\left(t_{b_l}, t_{e_l}\right)\right) = t_s\left(e_{ij}, l_1, Rs\left(t_{b_l}, t_{e_l}\right)\right) = \ldots = t_s\left(e_{ij}, l_k, Rs\left(t_{b_l}, t_{e_l}\right)\right)$$

$$t_f\left(e_{ij}, l_0, Rs\left(t_{b_l}, t_{e_l}\right)\right) = t_f\left(e_{ij}, l_1, Rs\left(t_{b_l}, t_{e_l}\right)\right) = \ldots = t_f\left(e_{ij}, l_k, Rs\left(t_{b_l}, t_{e_l}\right)\right)$$

The communication duration on the links of the route step is determined by the slowest link of the route step. Therefore, the finish time of edge e_{ij} on link l_k is given by

$$t_f\left(e_{ij}, l_k, Rs\left(t_{b_l}, t_{e_l}\right)\right) = t_s\left(e_{ij}, l_k, Rs\left(t_{b_l}, t_{e_l}\right)\right) + \frac{c\left(e_{ij}\right)}{\min\limits_{l_n}\left\{b\left(l_n\right)\right\}}$$

where $\min\limits_{l_n}\left\{b\left(l_n\right)\right\}$ is the minimum data rate of the links in the route step $Rs\left(t_{b_l}, t_{e_l}\right)$. Since e_{ij} has the same start/finish time on all the links of the route step, they are uniformly denoted by $t_s\left(e_{ij}, Rs\left(t_{b_l}, t_{e_l}\right)\right)$ and $t_f\left(e_{ij}, Rs\left(t_{b_l}, t_{e_l}\right)\right)$ and present respectively the start and finish time of e_{ij} on all the links of route step $Rs\left(t_{b_l}, t_{e_l}\right)$.

The communicator c_l is occupied by e_{ij} from the start time of the setup on the processor to the finish time on the links of the route step. Therefore, the start time of e_{ij} on c_l is given by $t_s\left(e_{ij}, c_l, Rs\left(t_{b_l}, t_{e_l}\right)\right) = t_s\left(e_{ij}, p_l, Rs\left(t_{b_l}, t_{e_l}\right)\right)$, and the finish time of e_{ij} on c_l is given by $t_f\left(e_{ij}, c_l, Rs\left(t_{b_l}, t_{e_l}\right)\right) = t_f\left(e_{ij}, Rs\left(t_{b_l}, t_{e_l}\right)\right)$.

Figure 3.6(a) shows the performing of communication with communicator that is configured by the beginning terminal P1. A communication may be delayed after the configuring because some links of the route step are occupied; therefore, the communication is held on the communicator and occupies a longer duration in this case. In addition, the communicator can also be configured by the ending terminal. Figure 3.6(b) shows a communication that is set up by the ending terminal and is delayed after the configuring.

Performing Communication by Processor

If a processor p_l is used to perform a communication e_{ij} on a route step $Rs\left(t_{b_l}, t_{e_l}\right)$, e_{ij} must be aligned on p_l and all the links of the route step $Rs\left(t_{b_l}, t_{e_l}\right)$. The processor does not need to configure a communicator, but it is occupied during the performing of the communication. Figure 3.7 shows the performing of a communication with processor that may be either the beginning terminal (Figure 3.7(a)) or the ending terminal (Figure 3.7(b)).

The start time of e_{ij} on p_l is same to the start time of e_{ij} on all the links of

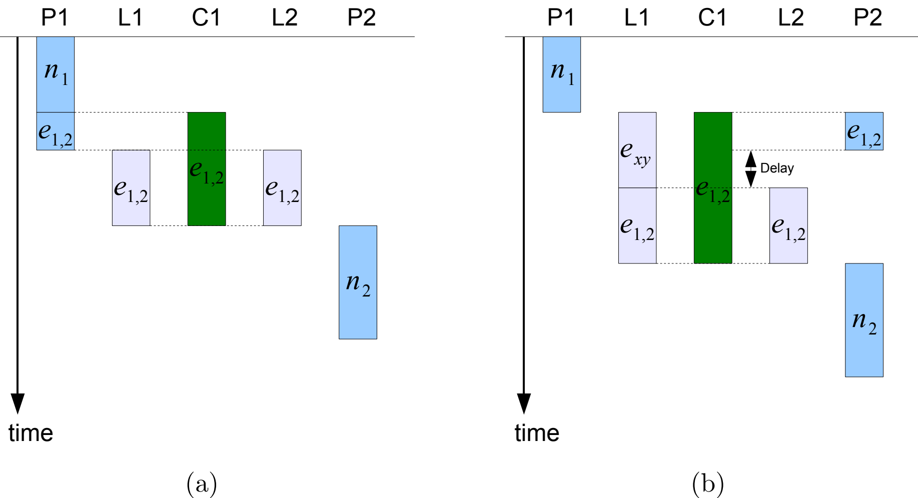


Figure 3.6: *Performing communication by communicator*

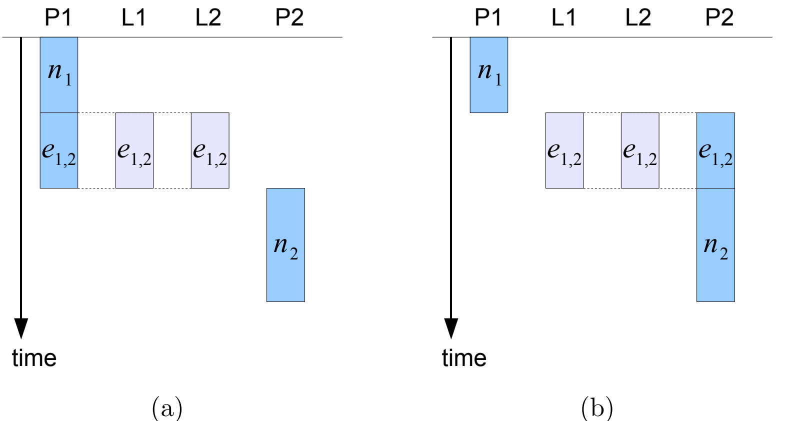


Figure 3.7: *Performing communication by processor*

$Rs(t_{b_l}, t_{e_l})$, that is

$$t_s(e_{ij}, p_l, Rs(t_{b_l}, t_{e_l})) = t_s(e_{ij}, Rs(t_{b_l}, t_{e_l}))$$

The finish time of e_{ij} on p_l is same to the finish time of e_{ij} on all the links of $Rs(t_{b_l}, t_{e_l})$ and is given by

$$t_f(e_{ij}, p_l, Rs(t_{b_l}, t_{e_l})) = t_f(e_{ij}, Rs(t_{b_l}, t_{e_l})) = t_s(e_{ij}, Rs(t_{b_l}, t_{e_l})) + \frac{c(e_{ij})}{\min\limits_{l_n}\{b(l_n)\}}$$

3.3.3 Causality Conditions

A computation can not be started on an operator when its input data are not available on this operator, and its output data can not be transferred to other operators when this computation is not finished. This is the general causality of task scheduling. Supposing an edge e_{ij} from n_i on $oper(n_i)$ to n_j on $oper(n_j)$, $oper(n_i) \neq oper(n_j)$, e_{ij} is assigned on the route $R = \langle Rs(t_{b_1}, t_{e_1}), \ldots, Rs(t_{b_l}, t_{e_l}), \ldots, Rs(t_{b_m}, t_{e_m})\rangle$ with $t_{b_1} = oper(n_i)$ and $t_{e_m} = oper(n_j)$. The causality also needs to satisfy the store-and-forward mode between route steps. Following inequalities describe the causality condition:

$$t_f(n_i, oper(n_i)) \leq t_s(e_{ij}, p_1, Rs(t_{b_1}, t_{e_1})) \tag{3.1}$$

$$\left.\begin{array}{r} t_f(e_{ij}, Rs(t_{b_1}, t_{e_1})) \leq t_s(e_{ij}, p_2, Rs(t_{b_2}, t_{e_2})) \\ \cdots \\ t_f(e_{ij}, Rs(t_{b_l}, t_{e_l})) \leq t_s\left(e_{ij}, p_{l+1}, Rs\left(t_{b_{l+1}}, t_{e_{l+1}}\right)\right) \\ \cdots \\ t_f\left(e_{ij}, Rs\left(t_{b_{m-1}}, t_{e_{m-1}}\right)\right) \leq t_s(e_{ij}, p_m, Rs(t_{b_m}, t_{e_m})) \end{array}\right\} \tag{3.2}$$

$$t_f(e_{ij}, Rs(t_{b_m}, t_{e_m})) \leq t_s(n_j, oper(n_j)) \tag{3.3}$$

Equation 3.1 means a communication should be started after its origin computation is finished. The store-and-forward routing on a route is described by Equation 3.2. Equation 3.3 means a computation should be started after its input communication is finished. If the communication of e_{ij} does not exist (i.e. $oper(n_i) = oper(n_j)$), the causality condition is simplified as $t_f(n_i, oper(n_i)) \leq t_s(n_j, oper(n_i))$.

The time when all its input communications are finished is called a computation's Data Ready Time (DRT) and is obtained by

$$\begin{aligned} DRT(n_j, oper(n_j)) &= \max\left\{ \max_{e_{ij} \in E, oper(n_i) \neq oper(n_j)} \{t_f(e_{ij}, Rs(t_{b_m}, oper(n_j)))\}, \right. \\ &\quad \left. \max_{e_{ij} \in E, oper(n_i) = oper(n_j)} \{t_f(n_i, oper(n_i))\} \right\} \end{aligned} \tag{3.4}$$

$Rs(t_{b_m}, oper(n_j))$ is the last route step for e_{ij} in Equation 3.4 if the communication is needed. DRT is the earliest time when a computation can be started. If n_j is a node without input edge, we have $DRT(n_j, op) = 0, \forall op \in OP$.

3.3.4 Scheduling Conditions

Computations and communications are inserted to the idle time intervals on operators, communicators and communication links during the scheduling. The following conditions should be satisfied to guarantee the causality during the insertion.

Node Scheduling Condition

For a node n_i, let $[A, B], (A, B) \in [0, \infty]^2$ be an idle time interval on the operator *op*. n_i can be scheduled on *op* within $[A, B]$ if $\max\{A, DRT(n_i, op)\} + w(n_i, op) \leq B$. The start time of n_i on *op* is given by $t_s(n_i, op) = \max\{A, DRT(n_i, op)\}$.

Edge Scheduling Condition with Communication Performed by Communicator

For an edge e_{ij} to be scheduled on the route step $Rs(t_{b_l}, t_{e_l})$, if the communication is performed by a communicator,

- let $[A_p, B_p], (A_p, B_p) \in [0, \infty]^2$ be an idle time interval on the processor p_l of $Rs(t_{b_l}, t_{e_l})$,
- let $[A_c, B_c], (A_c, B_c) \in [0, \infty]^2$ be an idle time interval on the communicator c_l of $Rs(t_{b_l}, t_{e_l})$,
- let $[A_l, B_l], (A_l, B_l) \in [0, \infty]^2$ be a common idle time interval on all the links of $Rs(t_{b_l}, t_{e_l})$.

We note $A = \max\{A_p, A_c\}$, $B = \min\{B_c, B_l\}$, and $C = \frac{c(e_{ij})}{\min\limits_{l_n}\{b(l_n)\}}$ where $\min\limits_{l_n}\{b(l_n)\}$ gives the minimum data rate of the links in the route step $Rs(t_{b_l}, t_{e_l})$. e_{ij} can be scheduled on this route step within $[A, B]$ if the following conditions are satisfied:

$$B_p \geq \begin{cases} \max\{A, t_f(n_i, oper(n_i))\} + s(p_l, c_l), \\ \quad \text{if } Rs(t_{b_l}, t_{e_l}) \text{ is the first route step} \\ \\ \max\left\{A, t_f\left(e_{ij}, Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)\right)\right\} + s(p_l, c_l), \\ \quad \text{otherwise} \end{cases}$$

$$B \geq \begin{cases} \max\left\{A_l, \max\left\{A, t_f\left(n_i, oper\left(n_i\right)\right)\right\} + s\left(p_l, c_l\right)\right\} + C, \\ \quad \text{if } Rs\left(t_{b_l}, t_{e_l}\right) \text{ is the first route step} \\ \\ \max\left\{A_l, \max\left\{A, t_f\left(e_{ij}, Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)\right)\right\} + s\left(p_l, c_l\right)\right\} + C, \\ \quad \text{otherwise} \end{cases}$$

where $s\left(p_l, c_l\right)$ is the setup time for p_l to configure c_l, and $Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)$ is the route step before $Rs\left(t_{b_l}, t_{e_l}\right)$.

The start time of e_{ij} on the processor p_l of $Rs\left(t_{b_l}, t_{e_l}\right)$ is given by

$$t_s\left(e_{ij}, p_l, Rs\left(t_{b_l}, t_{e_l}\right)\right) = \begin{cases} \max\left\{A, t_f\left(n_i, oper\left(n_i\right)\right)\right\}, \\ \quad \text{if } Rs\left(t_{b_l}, t_{e_l}\right) \text{ is the first route step} \\ \\ \max\left\{A, t_f\left(e_{ij}, Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)\right)\right\}, \\ \quad \text{otherwise} \end{cases}$$

The start time of e_{ij} on all the links of route step $Rs\left(t_{b_l}, t_{e_l}\right)$ is given by

$$t_s\left(e_{ij}, Rs\left(t_{b_l}, t_{e_l}\right)\right) = \begin{cases} \max\left\{A_l, \max\left\{A, t_f\left(n_i, oper\left(n_i\right)\right)\right\} + s\left(p_l, c_l\right)\right\}, \\ \quad \text{if } Rs\left(t_{b_l}, t_{e_l}\right) \text{ is the first route step} \\ \\ \max\left\{A_l, \max\left\{A, t_f\left(e_{ij}, Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)\right)\right\} + s\left(p_l, c_l\right)\right\}, \\ \quad \text{otherwise} \end{cases}$$

The start time of e_{ij} on all the links of the route step $Rs\left(t_{b_l}, t_{e_l}\right)$ satisfies the constraint condition of

$$t_s\left(e_{ij}, Rs\left(t_{b_l}, t_{e_l}\right)\right) \geq t_f\left(e_{ij}, p_l, Rs\left(t_{b_l}, t_{e_l}\right)\right)$$

It means a communication can not be started until the configuring is finished.

Edge Scheduling Condition with Communication Performed by Processor

For an edge e_{ij} to be scheduled on the route step $Rs\left(t_{b_l}, t_{e_l}\right)$, if the communication is performed by a processor, let $[A, B], (A, B) \in [0, \infty]^2$ be a common idle time interval on the processor p_l and all the links of $Rs\left(t_{b_l}, t_{e_l}\right)$. We note $C = \frac{c(e_{ij})}{\min\limits_{l_n}\{b(l_n)\}}$ where $\min\limits_{l_n}\left\{b\left(l_n\right)\right\}$ gives the minimum data rate of the links in the route step $Rs\left(t_{b_l}, t_{e_l}\right)$. e_{ij}

can be scheduled on this route step within $[A, B]$ if the following condition is satisfied:

$$B \geq \begin{cases} \max\{A, t_f(n_i, oper(n_i))\} + C, \\ \quad \text{if } Rs(t_{b_l}, t_{e_l}) \text{ is the first route step} \\ \\ \max\left\{A, t_f\left(e_{ij}, Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)\right)\right\} + C, \\ \quad \text{otherwise} \end{cases}$$

where $Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)$ is the route step before $Rs(t_{b_l}, t_{e_l})$.

The start time of e_{ij} on the processor p_l is same to the start time of e_{ij} on all the links of $Rs(t_{b_l}, t_{e_l})$ and is given by

$$t_s(e_{ij}, p_l, Rs(t_{b_l}, t_{e_l})) = t_s(e_{ij}, Rs(t_{b_l}, t_{e_l})) = \begin{cases} \max\{A, t_f(n_i, oper(n_i))\}, \\ \quad \text{if } Rs(t_{b_l}, t_{e_l}) \text{ is the first route step} \\ \\ \max\left\{A, t_f\left(e_{ij}, Rs\left(t_{b_{l-1}}, t_{e_{l-1}}\right)\right)\right\}, \\ \quad \text{otherwise} \end{cases}$$

3.4 Task Scheduling with Topology Graph Model

Since it is difficult to implement the task scheduling with the advanced architecture model at the very start, we begin with simplifying the architecture model and the task scheduling to research useful heuristic techniques. In fact, the architecture model is simplified to be the topology graph model, and the task scheduling becomes similar to be the task scheduling with communication contention in [SS05]. The work in the following chapters is to give advanced techniques to improve the performance of task scheduling based on the simplified task scheduling.

3.4.1 Topology Graph Model

An architecture of multiple processors interconnected by communication links and switches is modeled by a topology graph $TG = (N, P, D, H, b)$ in [SS05], where N is the set of vertices, P is a subset of N ($P \subseteq N$), D is the set of directed edges, H is the set of hyperedges, and b is the relative data rate of edge. The union of the two edge sets D and H is designated the link set L ($L = D \cup H$), and an element of this set is denoted by l ($l \in L$). Elements of these sets are explained as follows:

– a vertex $p \in P$ represents a processor,

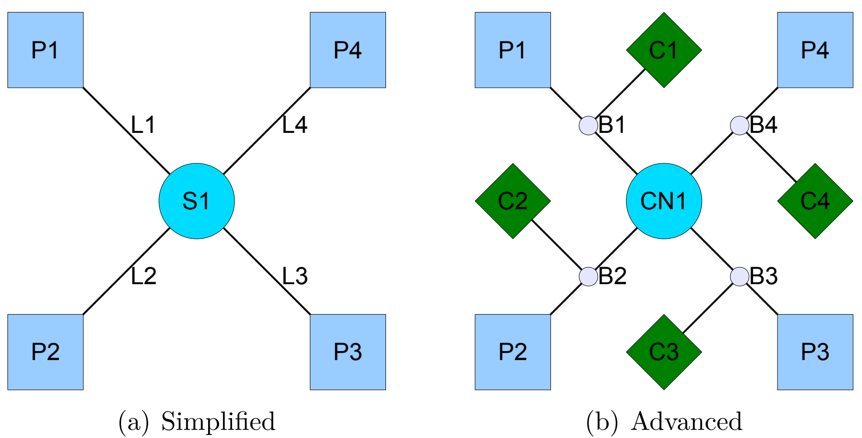
– a vertex $n \in N - P$ represents a switch,

– a directed edge $d \in D$ represents a directed communication link,

– a hyperedge $h \in H$ represents a multidirectional communication link.

A directed communication link usually represents a FIFO, and a multidirectional communication link that connects multiple vertices usually represents a half duplex bus. The positive weight $b(l)$, associated with a link $l \in L$, represents its relative data rate.

The topology graph model is a simplified model of our advanced architecture model. The processor, the switch, the directed communication link, and the multi-directional communication link in the topology graph model respectively correspond to the processor, the communication node, the bus and the FIFO in the advanced architecture model. IP coprocessor, communicator and memory are not modeled in the topology graph. Since a processor can be used to perform a communication when executing a computation in the topology graph model, the processor is equivalent to a processor with a communicator in the advanced architecture model. The processor can only use its communicator, and the setup time is neglected. Figure 3.8 compares a topology graph with 4 processors interconnected by a switch (Figure 3.8(a)) to its equivalent advanced architecture model (Figure 3.8(b)).



(a) Simplified (b) Advanced

Figure 3.8: *Comparison of two models*

We continue to simplify the model by only using homogeneous buses in a topology graph. Since directed edges are not used in the model, the topology graph can be denoted by $TG = (N, P, L, b)$ with $L = H$. Switches are ideal and contention-free as communication node in the advanced architecture model. Since communication links are considered homogeneous, data rates of different links are same, and the

relative data rate for all the links is 1 ($b(l) = 1, \forall l \in L$). However, processor can be heterogeneous, and a computation may need different execution times on different processors.

Though a route in the advanced architecture usually consists of several route steps, we suppose that there is at least a route step between any two processors in the topology graph. Therefore, a route in the topology graph is simplified to be a chain of links connected by switches and finally connects the origin processor to the destination processor. Figure 3.9 gives an example of 6 processors interconnected by buses and switches, and $L1 \rightarrow L7 \rightarrow L4$ is a route from $P1$ to $P4$ in this architecture.

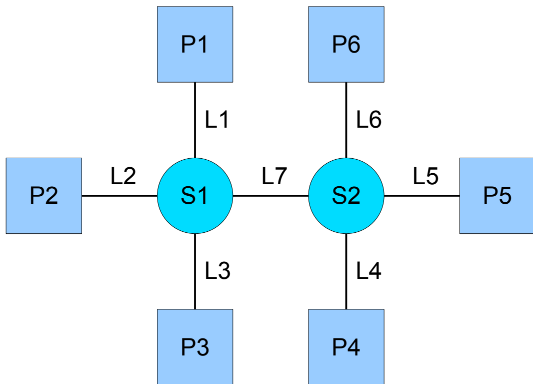


Figure 3.9: *Examples of architecture*

Routing is an important aspect of task scheduling. Since the scheduling is done at compile time, a route between two processors is also considered to be determined at compile time. Routes are determined once and stored in a table; therefore, the routing during the scheduling becomes looking up the table.

3.4.2 Scheduling with Communication Contention

Task scheduling with the topology graph model is a simplified version of the task scheduling with the advanced architecture model. It is a DAG scheduling with communication contention. A schedule of a DAG is the association of a start time and of a processor with each node of the DAG. When considering the communication contention, a schedule also includes allocating communications to links and associating start times on these links with each communication. A communication is aligned on all the links of a route in the cut-through mode. Therefore, it takes up the same duration on each link. However, a computation may take up different durations on

different processors because processors are heterogeneous.

A schedule S of a DAG $G = (V, E, w, c)$ over a topology graph $TG = (N, P, L, b)$ can be similarly described as in Section 3.3. We briefly give the difference as follows.

The start time of a node $n_i \in V$ on a processor $p \in P$ is denoted by $t_s(n_i, p)$; the finish time is given by $t_f(n_i, p) = t_s(n_i, p) + w(n_i, p)$, where $w(n_i, p)$ is the execution duration of n_i on p. Since we only use processor to execute computations in a topology graph, the set of processors on which n_i can be executed is denoted by $Proc(n_i)$, and the processor on which n_i is actually allocated is denoted by $proc(n_i)$. We use the average execution duration of a computation on different types of processors to represent the node weight which is similar to that for the advanced architecture model.

The finish time of a processor is the maximum finish time among all nodes allocated on this processor, $t_f(p) = \max_{proc(n_i)=p} \{t_f(n_i, proc(n_i))\}$; the schedule length of S is the maximum finish time among all the processors of the system, $sl(S) = \max_{p \in P} \{t_f(p)\}$.

The start time of an edge $e_{ij} \in E$ on a link $l \in L$ is denoted by $t_s(e_{ij}, l)$; the finish time of e_{ij} is given by $t_f(e_{ij}, l) = t_s(e_{ij}, l) + c(e_{ij})$.

The node scheduling condition for the topology graph is same to that for our advanced architecture model. However, the edge scheduling condition is a little different because a processor can perform a communication and a computation at the same time. In addition, since a route is simplified to be one-step, communications are handled in the way of cut-through on a route. Therefore, an edge e_{ij} is aligned on all the links of the route $l_{R_1} \to l_{R_2} \to \ldots \to l_{R_k}$, that is $t_s(e_{ij}, l_{R_1}) = t_s(e_{ij}, l_{R_2}) = \ldots = t_s(e_{ij}, l_{R_k})$ and $t_f(e_{ij}, l_{R_1}) = t_f(e_{ij}, l_{R_2}) = \ldots = t_f(e_{ij}, l_{R_k})$. The start time and finish time of e_{ij} on all the links of the route are denoted uniformly by $t_s(e_{ij})$ and $t_f(e_{ij})$ with $t_f(e_{ij}) = t_s(e_{ij}) + c(e_{ij})$.

Edge Scheduling Condition

For a DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$, let $l_{R_1} \to l_{R_2} \to \ldots \to l_{R_k}$ be route for an edge $e_{ij} \in E$ and let $[A, B], (A, B) \in [0, \infty]^2$ be a common idle time interval on all the links of the route. e_{ij} can be scheduled on this route within $[A, B]$ if $\max\{A, t_f(n_i, p)\} + c(e_{ij}) \le B$ where $p = proc(n_i)$. The start time of e_{ij} on this route is given by $t_s(e_{ij}) = \max\{A, t_f(n_i, p)\}$.

The topology graph model is not yet accurate enough because the processor involvement for communications is not taken into account. A new scheduling model is proposed based on the topology graph in [SSS06] to take into account the processor involvement for communications. However, the overhead and involvement for com-

munications on a processor are specified based on both the topology graph of target system and the DAG, and it complicates the specification. In addition, it is not practical that communicators and memories are not modeled but are assumed to be used in the communication subsystem of the topology graph for embedded systems.

3.5 Conclusion

This work concerns the static scheduling that is done at compile time and is more suitable for digital signal processing applications by leading to lower code size and higher computation efficiency. We firstly introduced the general task scheduling problem and gave a survey of the commonly used techniques for task scheduling. Then we formulated the task scheduling with our advanced architecture model. It consists in assigning computations and communications to components and finding time intervals on these components for the computations and communications. A computation is executed on an operator, which is a processor or an IP coprocessor. A communication is transferred on a route from one operator to another. A route usually contains several steps. Communications are handled in the cut-through mode on a route step and in the store-and-forward mode on different route steps. The start and finish times of computations and communications on different components were defined with the advanced architecture model. Based on the causality conditions with the architecture model, the scheduling conditions were finally given to be fulfilled during the scheduling.

Since it is difficult to start the research of task scheduling with the advanced architecture model, we simplified the advanced architecture model with a topology graph. Vertices of topology graph are processors and switches, and edges are communication links. Communications are assumed to be performed by a communication subsystem that is composed of switches and links and is contention-aware. The task scheduling problem is also simplified with the topology graph model and is indeed the task scheduling with communication contention. We will research advanced heuristics and techniques for the task scheduling with communication contention in the following chapters.

4

List Scheduling with Communication Contention

4.1 Introduction

Most scheduling heuristics are based on the approach of list scheduling presented in Chapter 3. This new chapter introduces list scheduling heuristics for the simplified task scheduling with communication contention for which the basic techniques have been given in [Sin07]. In addition to the two existing groups of node levels given in Chapter 2, we will explore three new groups of node levels for a DAG by considering the communication contention. All the five groups of node levels can be used as priorities to sort nodes for list scheduling. Using the five groups of node levels for the static and dynamic list scheduling usually leads to different scheduling orders of nodes and finally gives different scheduling results.

This chapter is organized as follows: Section 4.2 gives the three new groups of node levels considering the communication contention. Heuristics for static list scheduling and dynamic list scheduling are then given in Section 4.3. Section 4.4 gives experimental results, and the time complexities are analyzed in Section 4.5. This chapter is concluded in Section 4.6.

4.2 Node Levels with Communication Contention

Two groups of node levels have been defined in Chapter 2 and are named respectively as

– computation top level and bottom level (tl_{comp} and bl_{comp}),

– top level and bottom level (tl and bl).

These node levels have been used in task scheduling without communication contention. Since our work concerns about task scheduling with communication contention, it is necessary to consider the communication contention in the definition of node levels. Therefore, we propose three new groups of node levels by taking into account the input and output communication contention. These three groups are respectively named as

– input top level and bottom level (tl_{in} and bl_{in}),

– output top level and bottom level (tl_{out} and bl_{out}),

– input/output top level and bottom level (tl_{io} and bl_{io}).

Definitions of the new node levels are recursive like those of the existing node levels in Chapter 2. Figure 4.1 illustrates the dependency between nodes to define the new levels, where the dotted nodes and dotted edges are used to define the top levels and bottom levels of n_i.

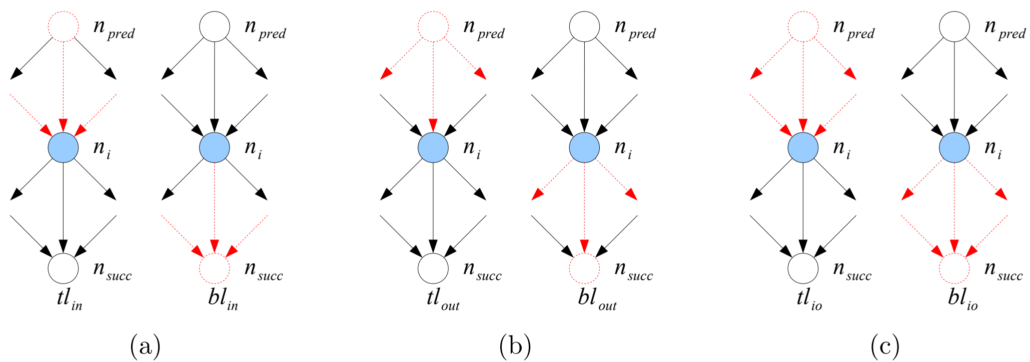


Figure 4.1: *Three new groups of node levels*

Given a DAG $G = (V, E, w, c)$, the formalized definitions of the three new top and bottom levels are given as follows:

Input top level and bottom level

As shown in Figure 4.1(a), the input top level and bottom level take into account the weights of nodes and the weights of all input edges of a node on the path. They

are defined recursively as

$$tl_{in}(n_i) = \begin{cases} 0, \text{if } n_i \in source(G) \\ \max\limits_{n_k \in pred(n_i)} \{tl_{in}(n_k) + w(n_k)\} + \sum\limits_{e_{li} \in E} c(e_{li}), \text{otherwise} \end{cases}$$

$$bl_{in}(n_i) = \begin{cases} w(n_i), \text{if } n_i \in sink(G) \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{in}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) \right\} + w(n_i), \text{otherwise} \end{cases}$$

Table 4.1 shows the input top level and bottom level of the DAG given in Figure 4.2.

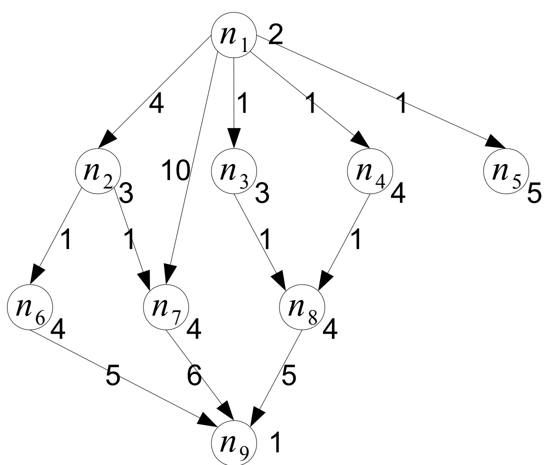


Figure 4.2: *A DAG example*

Table 4.1: *Input top level and bottom level*

n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
tl_{in}	0	6	3	3	3	11	20	9	40
bl_{in}	41	35	26	27	5	21	21	21	1
$tl_{in} + bl_{in}$	41	41	29	30	8	32	41	30	41

Output top level and bottom level

As shown in Figure 4.1(b), the output top level and bottom level take into account the weights of nodes and the weights of all output edges of a node on the path. They

are defined recursively as

$$tl_{out}(n_i) = \begin{cases} 0, \text{if } n_i \in source(G) \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{out}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) \right\}, \text{otherwise} \end{cases}$$

$$bl_{out}(n_i) = \begin{cases} w(n_i), \text{if } n_i \in sink(G) \\ \max\limits_{n_k \in succ(n_i)} \{ bl_{out}(n_k) \} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), \text{otherwise} \end{cases}$$

Table 4.2 shows the output top level and bottom level of the DAG given in Figure 4.2.

Table 4.2: *Output top level and bottom level*

n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
tl_{out}	0	19	19	19	19	24	24	24	34
bl_{out}	35	16	14	15	5	10	11	10	1
$tl_{out} + bl_{out}$	35	35	33	34	24	34	35	34	35

Input/output top level and bottom level

As shown in Figure 4.1(c), the input/output top level and bottom level take into account the weights of nodes and the weights of all input edges and all output edges of a node on the path. They are defined recursively as

$$tl_{io}(n_i) = \begin{cases} 0, \text{if } n_i \in source(G) \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{io}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) - c(e_{ki}) \right\} + \sum\limits_{e_{li} \in E} c(e_{li}), \text{otherwise} \end{cases}$$

$$bl_{io}(n_i) = \begin{cases} w(n_i), \text{if } n_i \in sink(G) \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{io}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) - c(e_{ki}) \right\} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), \text{otherwise} \end{cases}$$

Table 4.3 shows the input/output top level and bottom level of the DAG given in

Figure 4.2.

Table 4.3: *Input/output top level and bottom level*

n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
tl_{io}	0	19	19	19	19	24	34	25	54
bl_{io}	55	36	26	27	5	21	21	21	1
$tl_{io} + bl_{io}$	55	55	45	46	24	45	55	46	55

The new node levels can be computed by visiting each node in topological or inverse topological order as those for the existing node levels in Algorithm 2.3 and Algorithm 2.4 of Chapter 2. Though the items for *max* are more complicated according to the recursive definitions, the total repetition times are always at a level of $O(V + E)$, and the total complexity remains $O(V + E)$.

The new levels of a node can also be considered as communication contention path lengths of the longest path starting or ending at this node. Therefore, the sum of the new top level and bottom level of a node gives the communication contention path length of the longest path that passes this node in a DAG. A node with the maximum sum of its new top level and bottom level is on a critical path with communication contention. The critical paths with communication contention are respectively named as input critical path, as output critical path and as input/output critical path for the three new groups of node levels.

As shown in Table 4.1, the length of the input critical path is 41 for the DAG given in Figure 4.2. n_1, n_2, n_7 and n_9 are nodes on an input critical path. Similarly, the length of the output critical path is 35 (Table 4.2). n_1, n_2, n_7 and n_9 are nodes on an output critical path. The length of the input/output critical path is 55 (Table 4.3). n_1, n_2, n_7 and n_9 are nodes on an input/output critical path. Though the three critical path are same for this example, they are not necessarily same for any DAG.

Three nondecreasing orders of new top levels and three nonincreasing orders of new bottom levels are all topological orders. Table 4.4 gives the six node orders based on the six node levels with communication contention for the DAG in Figure 4.2. Nodes with the same level can be sorted randomly among themselves like $\{n_3, n_4, n_5\}$ for tl_{in}.

Table 4.4: *Different topological orders*

Node level	Node order
tl_{in}	$n_1, \{n_3, n_4, n_5\}, n_2, n_8, n_6, n_7, n_9$
bl_{in}	$n_1, n_2, n_4, n_3, \{n_6, n_7, n_8\}, n_5, n_9$
tl_{out}	$n_1, \{n_2, n_3, n_4, n_5\}, \{n_6, n_7, n_8\}, n_9$
bl_{out}	$n_1, n_2, n_4, n_3, n_7, \{n_6, n_8\}, n_5, n_9$
tl_{io}	$n_1, \{n_2, n_3, n_4, n_5\}, n_6, n_8, n_7, n_9$
bl_{io}	$n_1, n_2, n_4, n_3, \{n_6, n_7, n_8\}, n_5, n_9$

4.3 List Scheduling Heuristics

List scheduling is an important scheduling heuristic. It consists in firstly sorting nodes into a list and then scheduling each node of the list on a processor. List scheduling is classified as static or dynamic. This section gives the list scheduling heuristics in the case of communication contention.

4.3.1 Static List Scheduling Heuristic

Algorithm 4.1 is one static list scheduling heuristic. It is composed of three procedures of `Sort_Nodes()`, `Select_Processor()` and `Schedule_Node()`. Details of the three procedures are explained in the following sections.

Algorithm 4.1: **Static_List_Scheduling**(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule S of G on TG

1 $NodeList \leftarrow$ **Sort_Nodes**(V);
2 **for** *each* $n \in NodeList$ **do**
3 $\quad p_{best} \leftarrow$ **Select_Processor**(n, $Proc\,(n)$);
4 $\quad$ **Schedule_Node**(n, p_{best});
5 **end**

Sorting Nodes with Node Priorities

Nodes are firstly sorted into a list in the static list scheduling heuristic. The node order in the list should be a topological order because a node can only be exactly scheduled after its predecessors being scheduled. We usually have more opportunities to find an earlier time interval on a good processor when scheduling a node appearing earlier in the node list. As the number of scheduled nodes increases, it becomes

difficult to find earlier time intervals on processors, and the following nodes must be executed later. Since some nodes should be executed as early as possible to shorten the total execution time, the node order in the list affects much the schedule result, and it is important to get a good node order before scheduling.

A DAG usually has many different topological orders, but it is unpractical to test all the possibilities and choose the best one. Therefore, we need some general topological orders that usually give good schedule results. Node levels can be used as priorities to get different orders because nodes can be sorted into different topological orders by their levels. Since the bottom level of a node reflects the path length from this node to the end of a DAG, scheduling a node with greater bottom level as early as possible can make the total execution time shorter. Therefore, the bottom level is a good priority to sort nodes. In fact, list scheduling heuristics with different priority schemes to sort nodes have been compared in [SS04], and the experiments show that list scheduling with static list sorted by bottom level outperforms all other compared contention aware algorithms.

We also use the top level as an auxiliary priority when several nodes have the same bottom level. In fact, when two nodes have the same bottom level, the one with a longer path passing it is usually more "critical" than the other and therefore should be treated earlier. Since the sum of top level and bottom level gives the length of the longest path passing a node, the node with greater top level is a more "critical" one when their bottom levels are same. Nodes are sorted into a list of $NodeList$ in the procedure `Sort_Nodes()` according to the following rule:

- **Rule for Sorting Nodes**
  Nodes are sorted by the nonincreasing order of their bottom levels; if two nodes have equal bottom levels, the one with greater top level is placed before the other; if both the bottom level and the top level are equal, these nodes are sorted randomly.

All the five groups of node levels can be similarly used as node priorities according to this rule, and we usually get different node lists. Since the bottom level reflects the time needed from this node to the end of the graph, our bottom levels with communication contention reflect better the reality in the case of communication contention. However, it is uncertain which priority is better for a specific DAG. Therefore, we combine the five groups of priorities with the static list scheduling heuristic and choose the best result. The whole process is called the combined static list scheduling heuristic.

Processor Selection

List scheduling selects a good processor to execute a node. The processor allowing the earliest finish time of a node is selected for this node. The detail of selecting a processor for a node n_i is given in Algorithm 4.2.

Algorithm 4.2: **Select_Processor**(n_i, P)

Input: A node $n_i \in V$ and the set $Proc\,(n_i)$ of processors
Output: The best processor p_{best} for the input node n_i

```
1 Best_Finish_Time ← ∞;
2 for each p ∈ Proc(n_i) do
3 |   Finish_Time ← Schedule_Node(n_i, p);
4 |   if Finish_Time < Best_Finish_Time then
5 |   |   Best_Finish_Time ← Finish_Time;
6 |   |   p_best ← p;
7 |   end
8 end
```

Node Scheduling

A node is scheduled on a processor by respecting the node scheduling condition (cf. Section 3.3.4). The detail of scheduling a node n_i on a processor p is given in Algorithm 4.3.

Algorithm 4.3: **Schedule_Node**(n_i, p)

Input: $n_i \in V$ and a processor $p \in Proc\,(n_i)$
Output: The finish time of n_i on p

```
1 for each n_l ∈ pred(n_i), proc(n_l) ≠ p do
2 |   Schedule_Edge(e_li, p);
3 end
4 Calculate DRT of node n_i;
5 Find the earliest idle time interval for node n_i on processor p respecting the
  node scheduling condition;
6 Calculate the finish time of n_i on p;
```

Edge Scheduling

Algorithm 4.4 gives the method for edge scheduling by respecting the edge scheduling condition (cf. Section 3.4.2). Since an edge e_{ij} is scheduled only when its origin

node n_i has been scheduled, the scheduling of this edge needs additionally the processor p on which the destination node n_j of e_{ij} is to be scheduled.

Algorithm 4.4: **Schedule_Edge**(e_{ij}, p)

Input: $e_{ij} \in E$ and a processor $p \in Proc\,(n_j)$ on which the node n_j is to be scheduled
Output: None

```
1 if n_i is scheduled then
2 |   if proc(n_i) ≠ p then
3 |   |   Determine the route R from proc(n_i) to p by looking up the route table;
4 |   |   Find the earliest common idle time interval on all the links of R
  |   |   respecting the edge scheduling condition;
5 |   end
6 end
```

4.3.2 Dynamic List Scheduling Heuristic

Algorithm 4.5 gives the dynamic list scheduling heuristic. Since the node order is not determined before the scheduling but dynamically created during the scheduling, the procedure `Sort_Nodes()` for the static list scheduling heuristic is no longer necessary. We use a procedure `Choose_Node()` to choose a node to be scheduled. The procedures `Select_Processor()` and `Schedule_Node()` are same to those for the static list scheduling heuristic in Algorithm 4.1.

Algorithm 4.5: **Dynamic_List_Scheduling**(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

```
1 UnscheduledNodes ← V;
2 while UnscheduledNodes ≠ null do
3 |   n ← Choose_Node(UnscheduledNodes);
4 |   p_best ← Select_Processor(n, Proc(n));
5 |   Schedule_Node(n, p_best);
6 |   Remove n from UnscheduledNodes;
7 end
```

The dynamic node order must also be a topological order; therefore, a node to be scheduled must be a free node with all its predecessors being scheduled. The node chosen to be scheduled in the next step is one of the free nodes on a path with the

maximum length. Since the length of the longest path is crucial to the schedule length, this node must be treated immediately in order to be executed as soon as possible. This node is named the critical node and is obtained in Algorithm 4.6.

Algorithm 4.6: **Choose_Node**(UN)

Input: A set UN of all the unscheduled nodes
Output: The critical node n_c among all the unscheduled nodes

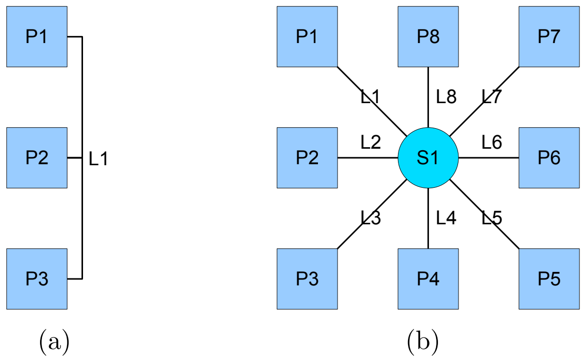
```
1  Create a set FN of all the free nodes from UN;
2  maxLength ← 0;
3  for each n_i ∈ FN do
4  |   length ← 0;
5  |   for each n_l ∈ pred(n_i) do
6  |   |   length ← max{length, t_f(n_l, proc(n_l)) + bl(n_i)};
7  |   end
8  |   if max_length < length then
9  |   |   max_length ← length;
10 |   |   n_c ← n_i;
11 |   else if max_length = length then
12 |   |   if bl(n_c) < bl(n_i) then
13 |   |   |   n_c ← n_i;
14 |   |   end
15 |   end
16 end
```

In this algorithm, the bottom level $bl(n_i)$ is the node priority. The bottom level reflects the time needed from this node to the end of the DAG, and our new bottom levels reflect better the reality in the case of communication contention. Therefore, $bl(n_i)$ can be replaced by other bottom levels like $bl_{comp}(n_i)$, $bl_{in}(n_i)$, $bl_{out}(n_i)$ and $bl_{io}(n_i)$. Different bottom levels may give different dynamic node orders and can finally lead to different schedule results. Similar to the static list scheduling, the dynamic list scheduling heuristic can be combined with the five bottom levels, and we choose the best result for a specific DAG. We call this whole process the combined dynamic list scheduling heuristic.

4.4 Experimental Results

This section compares the static and dynamic list scheduling heuristics with different node priorities. Figure 4.3 gives architectures that will be used in the following comparisons. Figure 4.3(a) is an architecture of three processors sharing a bus, and

Figure 4.3(b) gives another architecture of 8 processors connected to a switch by buses.



(a) (b)

Figure 4.3: *Examples of architecture*

4.4.1 Comparison with an Example

A DAG example is scheduled to show the performance of the list scheduling heuristics with different node priorities in this section. The DAG in Figure 4.2 is used as the algorithm, and the architecture is shown in Figure 4.3(a).

Static List Scheduling

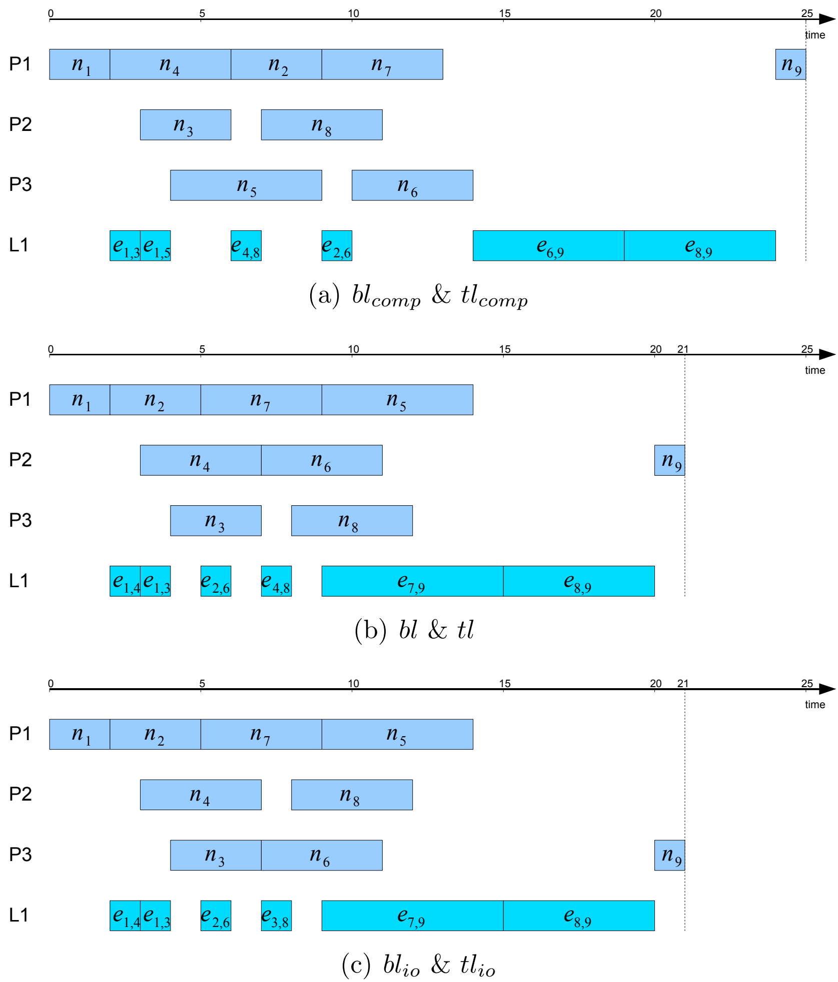
Table 4.5 gives the different node lists by using the five groups of node priorities for the DAG in Figure 4.2. When some nodes have the same priority, they are randomly sorted like $\{n_3, n_2\}$ for bl_{comp} & tl_{comp}.

Table 4.5: *Different static node lists*

Node priorities	Static node list
bl_{comp} & tl_{comp}	$n_1, n_4, \{n_3, n_2\}, n_8, \{n_7, n_6\}, n_5, n_9$
bl & tl	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$
bl_{in} & tl_{in}	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$
bl_{out} & tl_{out}	$n_1, n_2, n_4, n_3, n_7, \{n_6, n_8\}, n_5, n_9$
bl_{io} & tl_{io}	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$

The schedule result for the node list obtained by using bl_{comp} & tl_{comp} is shown in Figure 4.4(a) with the schedule length of 25. Since the node list obtained by using bl & tl is same to that obtained by using bl_{in} & tl_{in}, the same schedule result is obtained as in Figure 4.4(b) with the schedule length of 21. Using bl_{out} & tl_{out} can have the

same node list as that by using bl & tl and therefore gives the same schedule result. However, it can give another node list that is same as that by using bl_{io} & tl_{io}, and Figure 4.4(c) gives the schedule result with the schedule length of 21.



(a) bl_{comp} & tl_{comp}

(b) bl & tl

(c) bl_{io} & tl_{io}

Figure 4.4: *Schedule results of static heuristic with different node priorities*

Dynamic List Scheduling

The node list is generated dynamically during the scheduling of the dynamic heuristic. Table 4.6 gives the dynamic node lists by using different node priorities for the DAG in Figure 4.2.

The schedule result for the dynamic node list obtained by using bl_{comp} is shown in Figure 4.5(a) with the schedule length of 23. Since using bl, using bl_{in} and using bl_{out} give the same node list, the same schedule result is obtained as in Figure 4.5(b) with

Table 4.6: *Different dynamic node lists*

Node priorities	Dynamic node list
bl_{comp}	$n_1, n_4, n_3, n_8, n_2, n_6, n_7, n_9, n_5$
bl	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$
bl_{in}	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$
bl_{out}	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$
bl_{io}	$n_1, n_2, n_4, n_3, n_8, n_6, n_7, n_9, n_5$

the schedule length of 21. Figure 4.4(c) gives the schedule result by using bl_{io} with the schedule length of 18. This schedule result is better than all the others.

4.4.2 Comparison with Randomly Generated DAGs

We use random graphs to compare scheduling algorithms in order to get statistical results which are more persuasive than the result for a particular graph. In fact, we implemented a Java generator (1) based on SDF^3 graph random generator [SGB06] to generate random SDF graphs. The generated SDF graphs are constrained to have no cycle. Therefore, an SDF graph becomes a DAG without node and edge weights that will be generated randomly later.

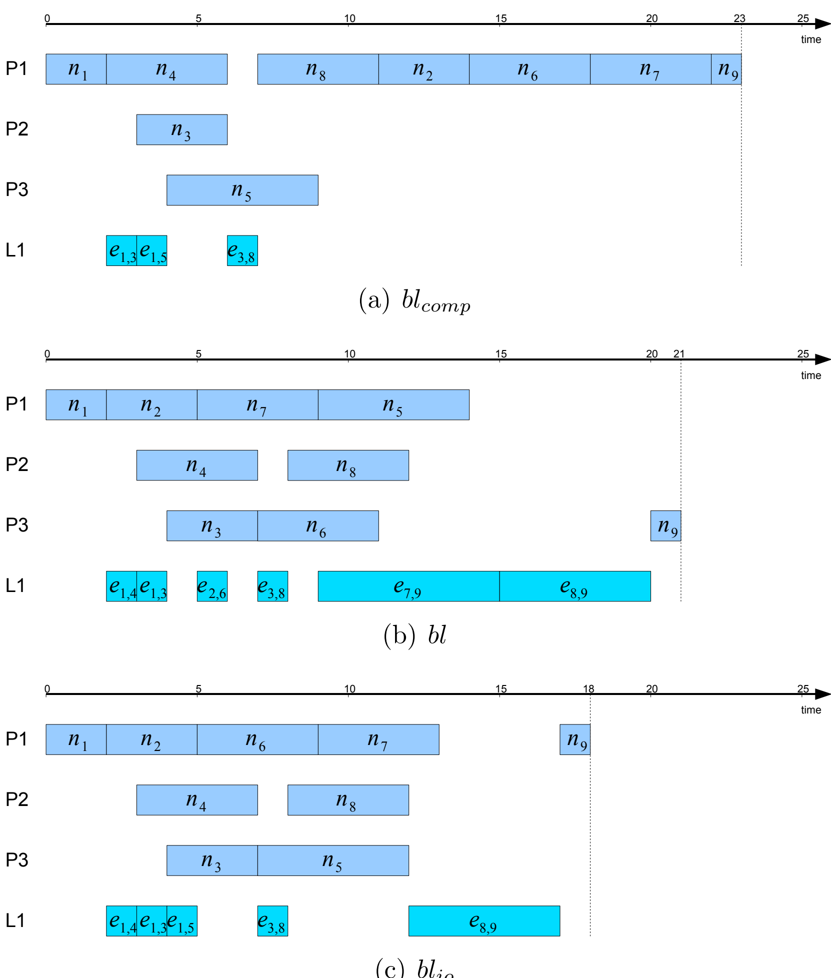
A random DAG is described in five aspects:

– the number of nodes,
– the average in degree,
– the average out degree,
– the random weights of nodes,
– the random weights of edges.

The average in degree and the average out degree are assumed to be same in our experiments, and the weights of nodes vary randomly from w_{min} to w_{max}. The communication to computation ratio (CCR) is used to generate random weights of edges. We define CCR as the average weight of edges divided by the average weight of nodes: $CCR = \frac{\frac{1}{|E|}\sum_{e \in E} c(e)}{\frac{1}{|V|}\sum_{n \in V} w(n)}$. Therefore, the weights of edges vary randomly from $w_{min} \times CCR$ to $w_{max} \times CCR$. The typical values of 0.1, 1 and 10 for CCR are used to respectively represent the low, medium and high communication cases and are tested in the following sections.

Since a list scheduling heuristic can use all the five groups of node priorities to

(1). http://sourceforge.net/projects/sdf4j



(a) bl_{comp}

(b) bl

(c) bl_{io}

Figure 4.5: *Schedule results of dynamic heuristic with different node priorities*

get different results, we combine the five groups of node priorities with a heuristic and choose the best result; the whole process is called a combined heuristic. The static list scheduling heuristic with nodes sorted by their bottom levels is used as the classic list scheduling heuristic, and we compare the schedule lengths of our list scheduling heuristics to that of the classic one. We define the acceleration factor (Acc) as $Acc = \frac{sl_{classic}}{sl_{compared}}$ to show the speed-up of the compared heuristic. We test 1000 random DAGs to obtain the statistical results for each group of DAG. The weights of nodes are generated randomly from $w_{min} = 100$ to $w_{max} = 1000$. The architecture given in Figure 4.3(b) is used for the following comparison.

Static List Scheduling

Figure 4.6 gives the average Acc of the static list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 0.1$. All the static heuristics have almost the same average Acc with $Acc \approx 1$. The average Acc of the combined static heuristic is improved only a little. In fact, the communication cost is very small in comparison with the computation cost, and it is difficult to obtain different node lists. Therefore, the schedule results of different node priorities are usually same, and the improvement on the average Acc of the combined static heuristic is not remarkable.

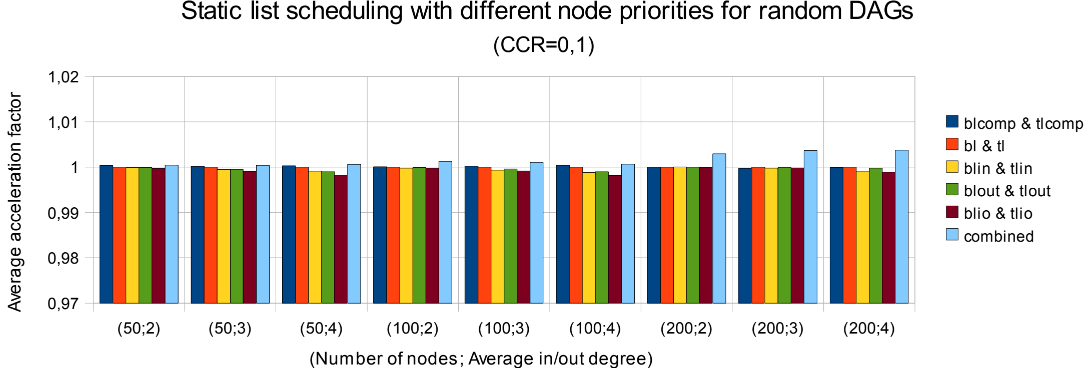


Figure 4.6: *Average Acc of static heuristic ($CCR = 0.1$)*

Figure 4.7 gives the average Acc of the static list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 1$. All the static heuristics have similar performances to the classic one ($Acc \approx 1$), but the combined static heuristic gives improvement on Acc. In fact, we may obtain different node lists with the five groups of node priorities when the communication cost increases. Though a static heuristic can not always give better schedule results than the classic one for a DAG, the combination of the five schedule results can usually improves the performance.

Figure 4.8 gives the average Acc of the static list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 10$. We can see that the static heuristic with *bl* & *tl*, which is essentially the classic list scheduling, usually gives greater Acc than the other four static heuristics. In fact, experiments in [SS04] have shown that this classic list scheduling usually outperforms other compared contention aware algorithms. However, our combined static list scheduling heuristic improves the final result because the classic one is not always

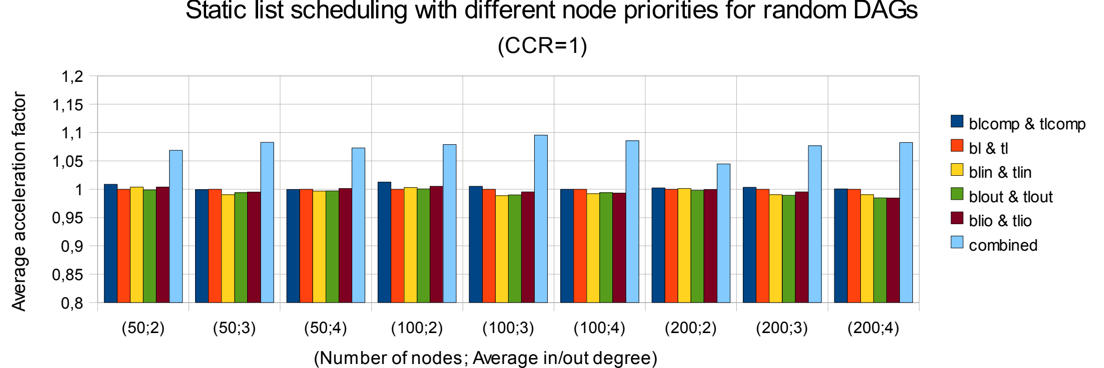


Figure 4.7: *Average Acc of static heuristic ($CCR = 1$)*

the best for any DAG.

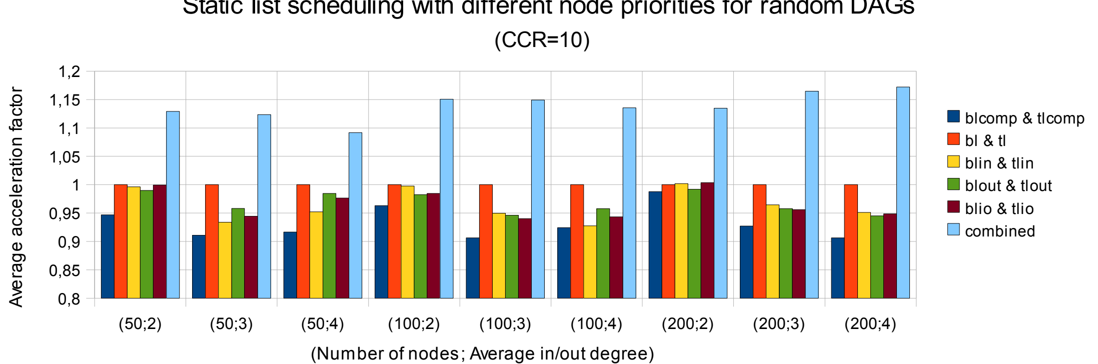


Figure 4.8: *Average Acc of static heuristic ($CCR = 10$)*

Figure 4.9 compares the average Acc of the combined static list scheduling heuristic for different CCR. We can see that the schedule results are obviously accelerated ($Acc > 1$) in the cases of $CCR = 1$ and $CCR = 10$ by using the combined static heuristic. The improvement increases when CCR varies from 0.1 to 10. If CCR is fixed, the combined static heuristic gives similar performances for DAGs of different sizes; therefore, the combined static heuristic is proved to be stable.

Dynamic List Scheduling

Figure 4.10 gives the average Acc of the dynamic list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 0.1$. All the dynamic heuristics give $Acc < 1$, and the Acc decreases as the number of nodes increases. As to the combined dynamic heuristic, we do not get any improvement

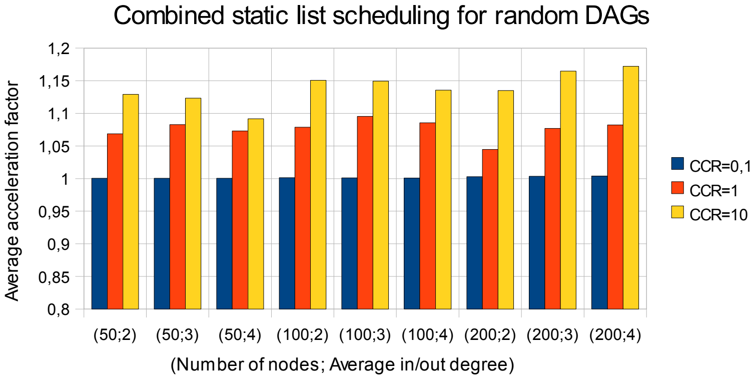


Figure 4.9: *Average Acc of combined static heuristic*

on average in comparison with the classic list scheduling. Therefore, the dynamic list scheduling heuristics can not give any improvement on average in the case of $CCR = 0.1$.

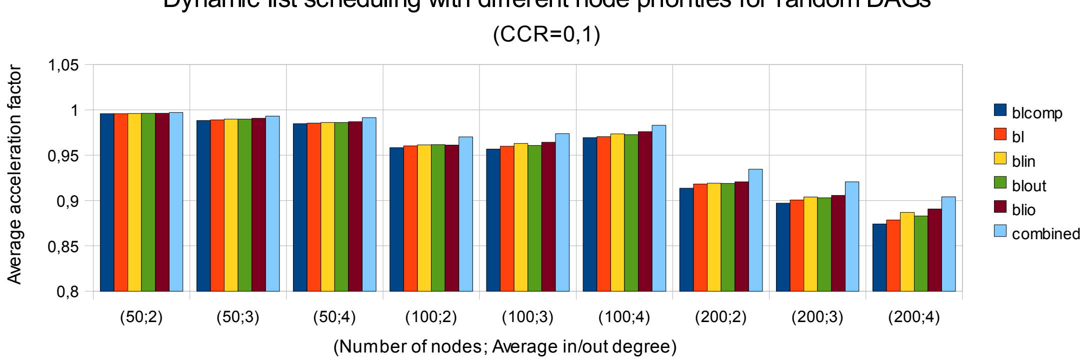


Figure 4.10: *Average Acc of dynamic heuristic ($CCR = 0.1$)*

Figure 4.11 gives the average Acc of the dynamic list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 1$. Though each dynamic heuristic has $Acc \leq 1$, the combined dynamic heuristic gives $Acc > 1$ when the number of nodes is not great (less than 200 in this figure). When DAGs become very complicated (e.g. have more than 200 nodes), the combined dynamic heuristic can not give any improvement on average.

Figure 4.12 gives the average Acc of the dynamic list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 10$. Though almost all the dynamic heuristics have $Acc < 1$, the combined dynamic heuristic stably gives $Acc > 1$. In fact, since the communication cost is much greater

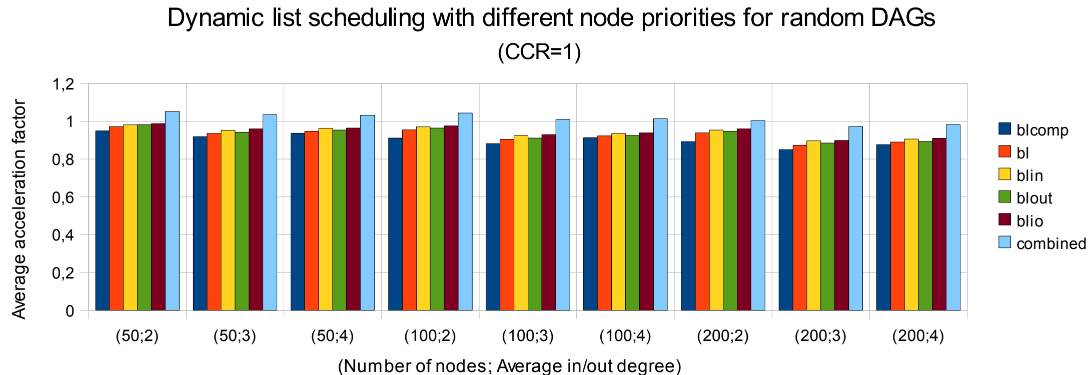


Figure 4.11: *Average Acc of dynamic heuristic ($CCR = 1$)*

than the computation cost, the node priority without considering the communication becomes unpractical and usually gives bad results. The other four node priorities with considering the communication give similar performances, and the combination of the different dynamic heuristics improves the final result.

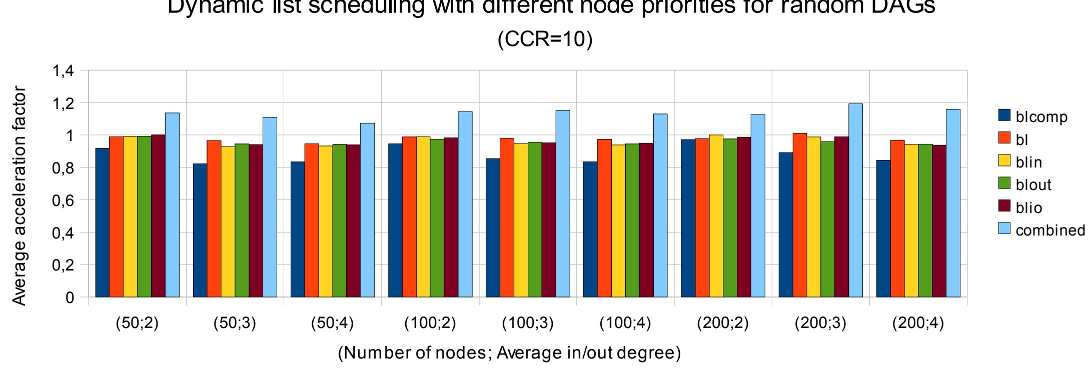


Figure 4.12: *Average Acc of dynamic heuristic ($CCR = 10$)*

Figure 4.13 compares the average Acc of the combined dynamic list scheduling heuristic for different values of CCR. Similar to the combined static list scheduling heuristic, the average Acc increases when CCR varies from 0.1 to 10 for each group of DAGs.

Static VS Dynamic

Figure 4.14 compares the combined static list scheduling heuristic to the combined dynamic list scheduling heuristic. The combined static heuristic usually has a greater Acc when $CCR = 0.1$ and $CCR = 1$, but its performance is close to that of the

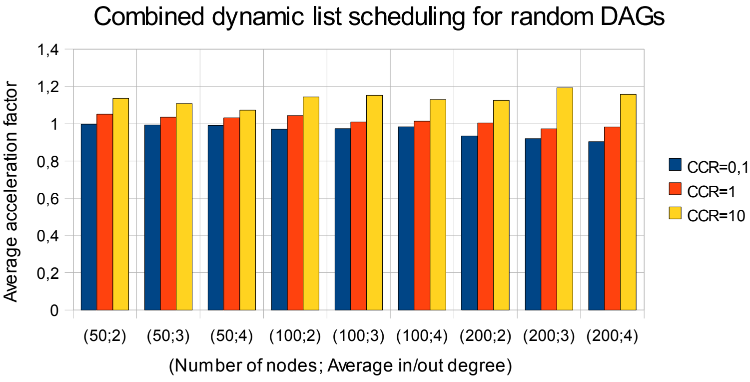


Figure 4.13: *Average Acc of combined dynamic heuristic*

dynamic heuristic for $CCR = 10$. Therefore, the best way should be using both the combined static and dynamic heuristics for a specific DAG and choosing the best schedule result.

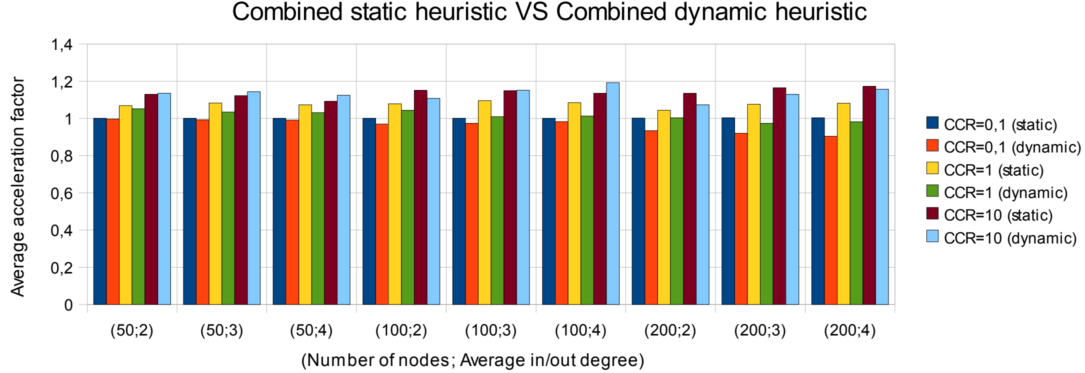


Figure 4.14: *Comparison of combined static heuristic to combined dynamic heuristic*

4.5 Analysis of Time Complexity

The time complexities of the list scheduling heuristics are analyzed as follows:

The route can be determined (calculated or looked up) in $O(1)$ time in the procedure `Schedule_Edge()` for static routing. If the route contains $O(routing)$ links, it takes $O(EO(routing))$ time to find the earliest common idle time interval on all links of the route. Thus, the complexity of `Schedule_Edge()` is $O(EO(routing))$.

The procedure `Schedule_Node()` needs firstly $O\left(\frac{E}{V}\right)$ times of `Schedule_Edge()` on average, then it takes $O\left(\frac{E}{V}\right)$ time to calculate the DRT, and it takes $O\left(\frac{V}{P}\right)$ time

to find an idle time interval for a node on average. At last, it takes $O(1)$ time to calculate the finish time of the node. Therefore, the total complexity of the procedure `Schedule_Node()` is $O\left(\frac{E^2O(routing)}{V} + \frac{V}{P}\right)$ on average.

As to the procedure `Select_Processor()`, it mainly calls $O(P)$ times of the procedure `Schedule_Node()`. Therefore, the complexity is $O\left(\frac{PE^2O(routing)}{V} + V\right)$.

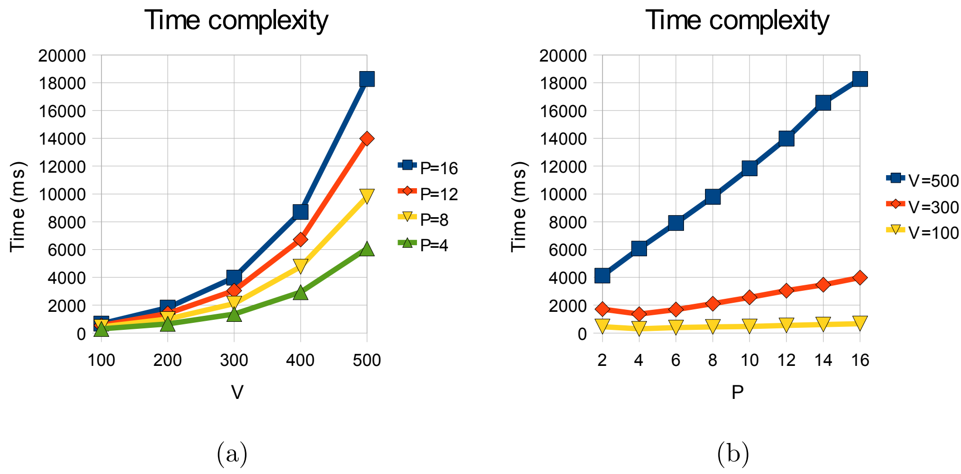
In Algorithm 4.1, sorting nodes has the complexity of $O(V \log V + E)$ (computing node levels in $O(V + E)$ + sorting in $O(V \log V)$). Our new definitions of top level and bottom level do not change the complexity of computing node levels; therefore, the complexity of sorting nodes is always $O(V \log V + E)$. Since the procedure `Select_Processor()` is more complicated than the procedure `Schedule_Node()`, the complexity in the for-loop is equal to that of the procedure `Select_Processor()`. Therefore, the total complexity is $O(PE^2O(routing) + V^2)$ for the static list scheduling heuristic.

Algorithm 4.5 consists of a procedure `Choose_Node()` in addition to the two procedures of `Select_Processor()` and `Schedule_Node()`. In `Choose_Node()`, creating the set of all the free nodes need to visit each node and has a complexity of $O(V)$. Then the total repetition times are no more than $V + E$ in the for-loop; therefore, the total complexity of the procedure `Choose_Node()` is $O(V + E)$. Since the procedure `Select_Processor()` is more complicated than the procedure `Schedule_Node()` and the procedure `Choose_Node()`, the complexity in the while-loop is equal to that of the procedure `Select_Processor()`. Therefore, the total complexity of the dynamic list scheduling heuristic is $O(PE^2O(routing) + V^2)$, which is of the same degree as that of the static list scheduling heuristic.

The degree of the time complexity for a combined list scheduling heuristic is not increased though it consists of five heuristics with different node priorities. Therefore, the time complexity of combined heuristics is also $O(PE^2O(routing) + V^2)$.

Figure 4.15 shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined static heuristic. All the DAGs have the average in/out degree of 4, and all the processors are connected to a switch. As shown in Figure 4.15(a) and Figure 4.15(b), the time increases with the square of V and increases linearly with P. We run our heuristic on a Pentium Dual-Core PC at 2.4GHz, and it takes about 18 seconds to schedule a DAG with 500 nodes on an architecture of 16 processors.

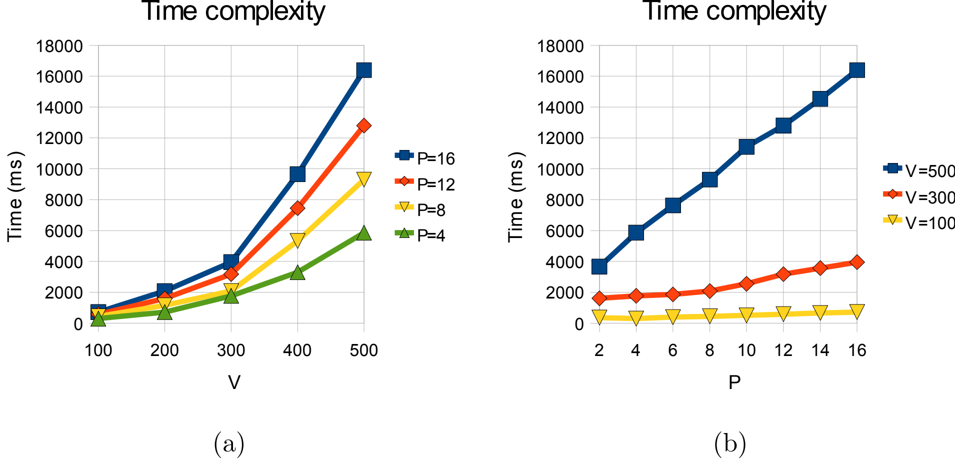
Figure 4.16 shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined dynamic heuristic. The result



(a) (b)

Figure 4.15: *Time complexity of static heuristic*

is similar to that of the combined static heuristic. The time increases with the square of V (Figure 4.16(a)) and increases linearly with P (Figure 4.16(b)). It takes about 16 seconds to schedule a DAG with 500 nodes on an architecture of 16 processors.



(a) (b)

Figure 4.16: *Time complexity of dynamic heuristic*

4.6 Conclusion

This chapter explored methods for the simplified task scheduling with communication contention, which used the DAG and topology graph to respectively model the algorithm and architecture. In addition to the two existing groups of node levels

for a DAG, we proposed three new groups of node levels by taking into account the communication contention. These new node levels can be similarly computed as the existing ones. All these node levels have similar properties and can be used to sort nodes in topological order.

As we said previously, the general task scheduling problem has been proven to be NP-hard, and many works try to find heuristics to go up to the optimal solution. Most heuristics are based on the approach of list scheduling. List scheduling is done at compile time. It can be classified as static or dynamic according to whether the node list is generated statically before the scheduling or dynamically during the scheduling. Using the five groups of node levels as priorities for the static and dynamic list scheduling usually leads to different scheduling orders of nodes and finally can give different scheduling results.

Since a list scheduling heuristic can not always give good schedule results for each DAG, we combined the five groups of node levels with the static and dynamic list scheduling heuristics and chose the best schedule result for a DAG. Experiments show that the combined static list scheduling heuristics always get improvements in comparison with the classic one for all kinds of random DAGs. As to the combined dynamic list scheduling, the performance is not improved in the case of $CCR = 0.1$; it is improved a little in the case of $CCR = 1$; but it is obviously improved in the case of $CCR = 10$. The improvement increases with the communication to computation ratio for both the combined static and dynamic heuristics. When comparing the combined static heuristic to the combined dynamic heuristic, it is difficult to say which one is better; therefore, the best way should be using both the combined static and dynamic heuristics for a specific DAG and choosing the best schedule result.

5

Advanced List Scheduling Methods

5.1 Introduction

Basic methods for list scheduling with communication contention have been given in Chapter 4. We have explored three new groups of node levels which are used together with the other two existing groups to compose combined heuristics. However, the improvement of the schedule result is not very great. In this chapter, we will give two advanced techniques to greatly improve the performance. The first technique uses the ***critical child*** of a node to select a processor for this node. The second technique called ***communication delay*** delays a communication when necessary to enlarge idle time intervals on communication links. These two techniques are used together to compose advanced list scheduling heuristics. The advanced list scheduling heuristics are also combined with different node priorities as proposed in Chapter 4 in order to improve the scheduling performance.

The rest of this chapter is organized as follows: Section 5.2 introduces the first technique of critical child to select a processor for a node. Then the second technique of communication delay for node and edge scheduling is given in Section 5.3. Heuristics for advanced static list scheduling and dynamic list scheduling are given in Section 5.4. Section 5.5 gives experimental results, and the time complexities of the advanced heuristics are analyzed in Section 5.6. This chapter is concluded in Section 5.7.

5.2 Processor Selection with Critical Child

Classic list scheduling heuristics select the processor allowing the earliest finish time for a node. This rule gives probably a locally optimized result. In fact, this rule usually gives bad results for the join structure of a DAG especially in the case of great communication cost and communication contention. Figure 5.1(a) shows such an example and Figure 5.1(b) gives the schedule result with the classic processor selection method, which selects a new processor for each one of n_1, n_2 and n_3 to provide the earliest finish time. Therefore, the execution of node n_4 has to wait until the communications from n_2 and n_3 finish, and the schedule length is 6 at last. By contrast, the schedule of all nodes on the same processor is shown in Figure 5.1(c) and has a schedule length of 4.

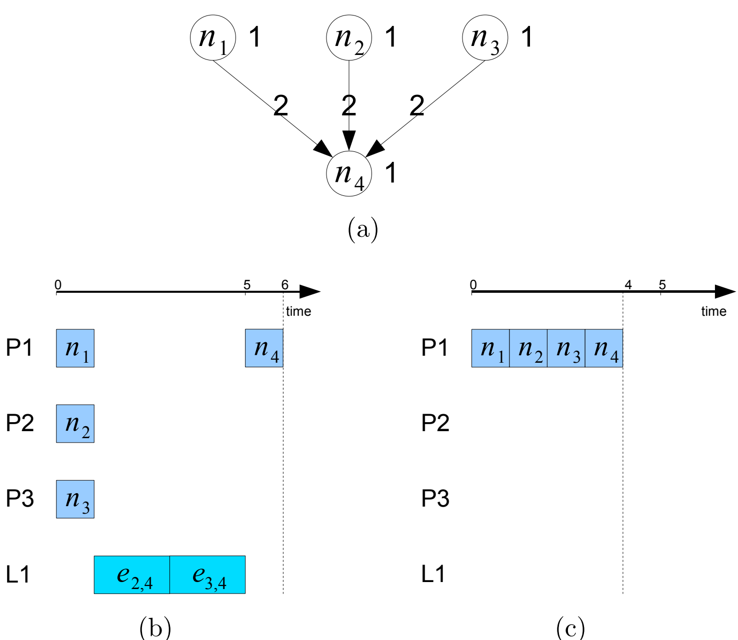


Figure 5.1: *A join DAG and two different schedule results*

In [KA96], the critical child of a node is defined as one of its successors that has the smallest difference between the absolute latest possible start time (ALST) and the absolute earliest possible start time (AEST). Then the critical child is used for scheduling with unbounded number of processors. We will use the concept of critical child for list scheduling with bounded number of processors in the case of communication contention. We redefine the critical child as follows.

Critical Child

Given a static node list $NodeList$, the critical child of node n_i is denoted by $cc(n_i)$ and is one of n_i's successors that firstly emerges in $NodeList$.

According to our definition, the critical child of n_i may be different if $NodeList$ differs. This is the difference between our critical child and that in [KA96].

Using critical child makes the processor selection take into account not only the predecessors of the node, but also its most important successor. Our method of using the critical child to select processor is given in Algorithm 5.1. Since it is possible that $cc(n_i)$ is not a free node with all its predecessors scheduled during the procedure of `Select_Processor()`, the scheduling of $cc(n_i)$ takes into account merely its scheduled predecessors and the edges between $cc(n_i)$ and them.

Algorithm 5.1: **Select_Processor**(n_i, P)

Input: A node $n_i \in V$ and the set P of all processors
Output: The best processor p_{best} for the input node n_i

```
1  Choose the critical child cc(n_i);
2  BestFinishTime ← ∞;
3  for each p ∈ Proc(n_i) do
4  |   FinishTime ← Schedule_Node(n_i, p, true);
5  |   MinFinishTime ← ∞;
6  |   if cc(n_i) ≠ null then
7  |   |   for each p' ∈ Proc(cc(n_i)) do
8  |   |   |   FinishTime ← Schedule_Node(cc(n_i), p', true);
9  |   |   |   MinFinishTime ← min{MinFinishTime, FinishTime};
10 |   |   end
11 |   else
12 |   |   MinFinishTime ← FinishTime;
13 |   end
14 |   if MinFinishTime < BestFinishTime then
15 |   |   BestFinishTime ← MinFinishTime;
16 |   |   p_best ← p;
17 |   end
18 end
```

5.3 Node and Edge Scheduling with Communication Delay

Our methods of node and edge scheduling differ from those given in Chapter 4 by using the As Late As Possible (ALAP) start time to delay communications. Given the route $l_{R_1} \rightarrow l_{R_2} \rightarrow \ldots \rightarrow l_{R_k}$ for edge e_{ij}, let e_{R_m} be the edge before which e_{ij} is scheduled on link l_{R_m}, the ALAP for edge e_{ij} is defined as

$$ALAP\left(e_{ij}\right) = \min\left\{t_s\left(e_{R_1}\right), t_s\left(e_{R_2}\right), \ldots, t_s\left(e_{R_k}\right), t_s\left(n_j, proc\left(n_j\right)\right)\right\} - c\left(e_{ij}\right)$$

where $t_s\left(e_{R_m}\right) = \infty$ if e_{ij} is the last edge scheduled on l_{R_m}.

The communication can be delayed by using the ALAP, and an idle time interval is therefore enlarged on a link. The idle time interval between two successive edges e_{n-1} and e_n on a link l changes from $\left[t_f\left(e_{n-1}, l\right), t_s\left(e_n, l\right)\right]$ to $\left[t_f\left(e_{n-1}, l\right), ALAP\left(e_n\right)\right]$, where $t_f\left(e_{n-1}, l\right) = 0$ if e_n is the first edge on link l and $t_s\left(e_n, l\right) = ALAP\left(e_n\right) = \infty$ if e_{n-1} is the last edge on link l. Figure 5.2 gives an example to show the use of ALAP. If e_{ij} is delayed to its ALAP, the idel time interval on $L1$ between e_{ab} and e_{ij} will be enlarged.

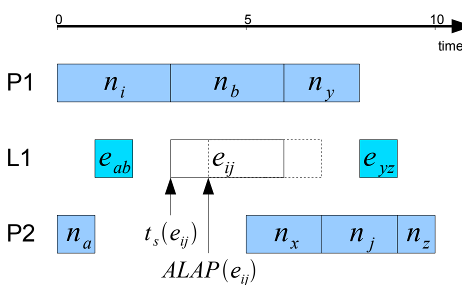


Figure 5.2: *Communication delay*

5.3.1 Node Scheduling

The method of scheduling a node n_i onto a processor p is given in Algorithm 5.2. When a node is scheduled, the ALAPs of its input edges are then calculated (line 6 to 10 in Algorithm 5.2). The ALAP of an edge can not be calculated during the processor selection. Therefore, a Boolean value is used to indicate whether the procedure `Schedule_Node()` is used in the procedure `Select_Processor()` or not.

Algorithm 5.2: **Schedule_Node**(n_i, p, $isTemporary$)

Input: $n_i \in V$, a processor $p \in P$ and a Boolean value $isTemporary$ indicating whether or not a temporary try for selecting processor
Output: The finish time of n_i on p

```
1  for each n_l ∈ pred(n_i), proc(n_l) ≠ p do
2  |   Schedule_Edge(e_li, p);
3  end
4  Calculate DRT of node n_i;
5  Find the earliest idle time interval for node n_i on processor p respecting the
   node scheduling condition;
6  if isTemporary = false then
7  |   for each n_l ∈ pred(n_i), proc(n_l) ≠ p do
8  |   |   Calculate the ALAP of e_li;
9  |   end
10 end
11 Calculate the finish time of n_i on p;
```

5.3.2 Edge Scheduling

Since an edge is scheduled only when its origin node has been scheduled, the scheduling of this edge needs additionally the processor on which the destination node of this edge will be scheduled. Algorithm 5.3 gives the method for edge scheduling. This algorithm is similar to that given in Chapter 4 except that the ALAP is considered in the edge scheduling condition (cf. Section 3.4.2).

Algorithm 5.3: **Schedule_Edge**(e_{ij}, p)

Input: $e_{ij} \in E$ and a processor $p \in P$ on which the node n_j is to be scheduled
Output: None

```
1  if n_i is scheduled then
2  |   if proc(n_i) ≠ p then
3  |   |   Determine the route R from proc(n_i) to p by looking up the route table;
4  |   |   Find the earliest common idle time interval on all the links of R
   |   |   respecting the edge scheduling condition with ALAP;
5  |   end
6  end
```

Figure 5.3 gives a DAG example to show the effect of communication delay. Nodes are sorted into a static order of n_1, n_2, n_3, n_4, n_5, n_6 by using the priority of *bl* & *tl*. Figure 5.4(a) gives a partial schedule result with n_1, n_2, n_3, n_4 having been scheduled. As to n_5, the input edge $e_{1,4}$ for n_4 can start at its ALAP of time 3. Therefore, the

edge $e_{1,5}$ is inserted between $e_{1,3}$ and $e_{1,4}$ as shown in Figure 5.4(b) and finally a schedule length of 8 is obtained in Figure 5.4(c). If ALAP is not used, a different schedule result is obtained in Figure 5.4(d) with the schedule length of 9.

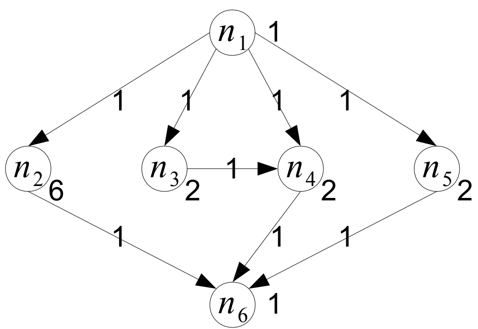
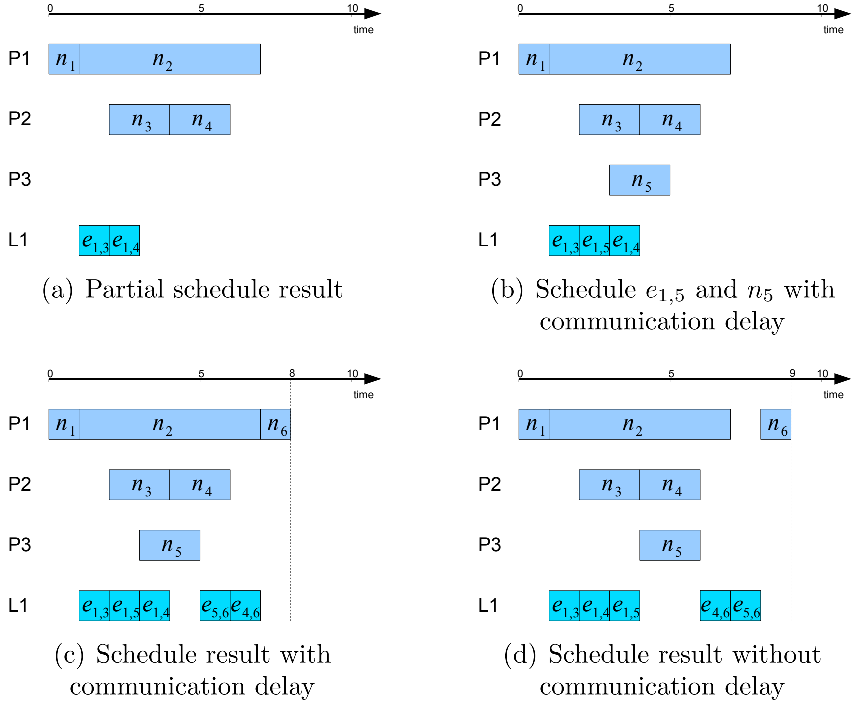


Figure 5.3: *A DAG example*



(a) Partial schedule result

(b) Schedule $e_{1,5}$ and n_5 with communication delay

(c) Schedule result with communication delay

(d) Schedule result without communication delay

Figure 5.4: *Scheduling procedures with/without communication delay*

5.4 Advanced List Scheduling Heuristics

We call the list scheduling heuristics with the techniques given above the advanced list scheduling heuristics. By contrast, the list scheduling heurisitcs given in Chapter 4

are called classic list scheduling heuristics. The advanced list scheduling heuristics differ only a little from the classic list scheduling heuristics by using a Boolean value in the procedure `Schedule_Node()`. Algorithm 5.4 and Algorithm 5.5 respectively give our advanced static list scheduling heuristic and advanced dynamic list scheduling heuristic.

Algorithm 5.4: **Advanced_Static_List_Scheduling**(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

1 $NodeList \leftarrow$ **Sort_Nodes**(V);
2 **for** *each* $n \in NodeList$ **do**
3     $p_{best} \leftarrow$ **Select_Processor**(n, P);
4   **Schedule_Node**(n, p_{best}, $false$);
5 **end**

Algorithm 5.5: **Advanced_Dynamic_List_Scheduling**(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

1 $UnscheduledNodes \leftarrow V$;
2 **while** $UnscheduledNodes \neq null$ **do**
3     $n \leftarrow$ **Choose_Node**($UnscheduledNodes$);
4     $p_{best} \leftarrow$ **Select_Processor**(n, P);
5   **Schedule_Node**(n, p_{best}, $false$);
6   Remove n from $UnscheduledNodes$;
7 **end**

5.5 Experimental Results

This section compares the advanced list scheduling heuristics with the classic ones. Figure 5.5 gives architectures that will be used in the following comparisons. Figure 5.5(a) is an architecture of three processors sharing a bus, and Figure 5.5(b) gives another architecture of 8 processors connected to a switch by buses. Experimental results are given as follows.

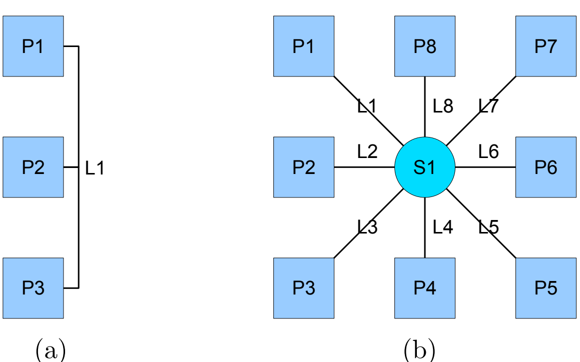


Figure 5.5: *Examples of architecture*

5.5.1 Comparison with an Example

As in Section 4.4.1, we use an example to show the performance of the advanced list scheduling heuristics with different node priorities. The DAG in Figure 5.6 is used as the algorithm, and the architecture is shown in Figure 5.5(a).

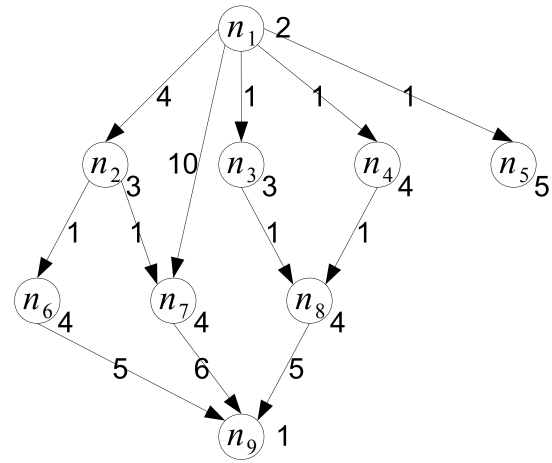


Figure 5.6: *A DAG example*

Advanced Static List Scheduling

Since the node list for static list scheduling heuristics only depends on the node priority, node lists for the advanced static list scheduling heuristic are same to those for the classic static list scheduling heutistic, and they are shown in Table 5.1 by using the five groups of node priorities. Table 5.2 gives the critical child for each node according to the static node lists in Table 5.1.

Table 5.1: *Different static node lists*

Node priorities	Static node list
bl_{comp} & tl_{comp}	$n_1, n_4, \{n_3, n_2\}, n_8, \{n_7, n_6\}, n_5, n_9$
bl & tl	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$
bl_{in} & tl_{in}	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$
bl_{out} & tl_{out}	$n_1, n_2, n_4, n_3, n_7, \{n_6, n_8\}, n_5, n_9$
bl_{io} & tl_{io}	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$

Table 5.2: *Critical children according to different node priorities*

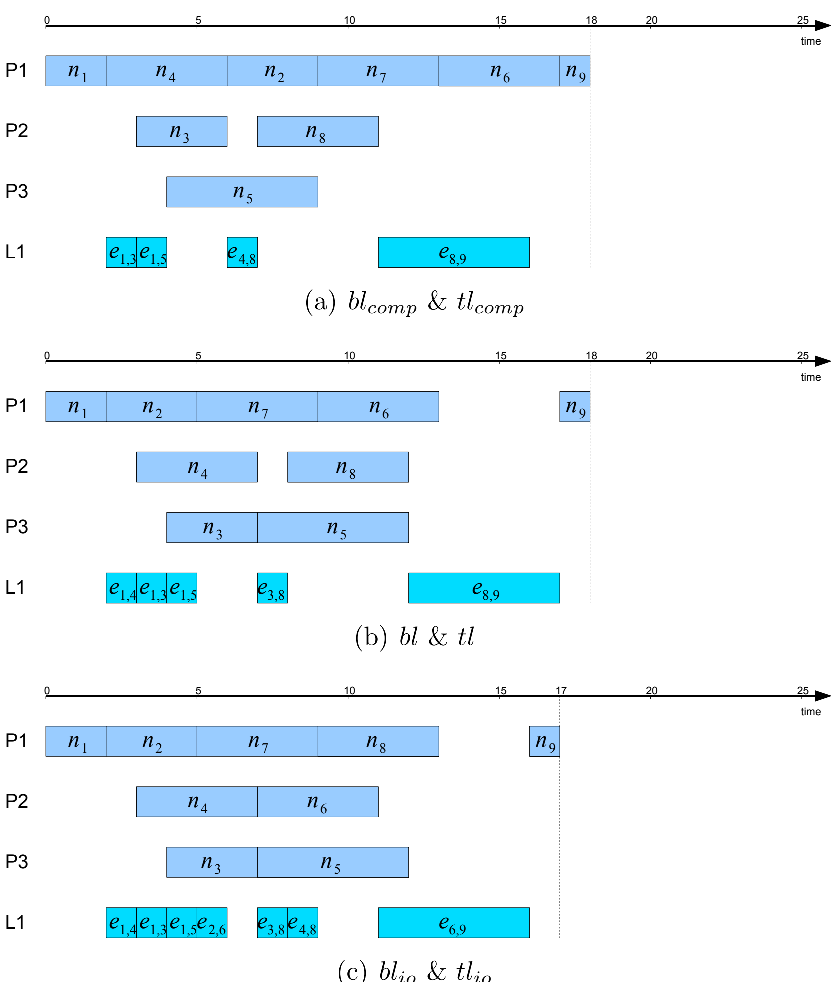
n_i	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
bl_{comp} & tl_{comp}	n_4	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl & tl	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl_{in} & tl_{in}	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl_{out} & tl_{out}	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
bl_{io} & tl_{io}	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null

The effect of the advanced static list scheduling heuristic with different priorities to sort nodes is given as follows. The schedule result for the node list sorted by bl_{comp} & tl_{comp} is shown in Figure 5.7(a) with the schedule length of 18. Since the node list is the same by bl & tl and by bl_{in} & tl_{in}, the same schedule result is obtained as in Figure 5.7(b) with the schedule length of 18. Figure 5.7(c) gives the schedule result for the same node list sorted by bl_{out} & tl_{out} and by bl_{io} & tl_{io}. A schedule length of 17 is obtained and is better than the two schedule lengths of 18. By contrast, the classic static list scheduling heuristics obtain three schedule results with schedule lengths of 25, 21 and 21 in Figure 4.4. Therefore, the advanced static list scheduling heuristics are better than the classic ones.

Advanced Dynamic List Scheduling

Since a dynamic node list is generated during the scheduling, we can not use the dynamic node list but use the corresponding static node list to determine the critical child for each node as shown in Table 5.2. The dynamic node list depends on the dynamic heuristic. Table 5.3 gives the node orders of the advanced dynamic heuristic by using different node priorities for the DAG in Figure 5.6.

The schedule result for the node priority bl_{comp} is shown in Figure 5.8(a) with the schedule length of 18. Since the node list is the same by bl, by bl_{out} and by bl_{io}, the same schedule result is obtained as in Figure 5.8(b) with the schedule length of 17.



(a) bl_{comp} & tl_{comp}

(b) bl & tl

(c) bl_{io} & tl_{io}

Figure 5.7: *Schedule results of advanced static heuristic with different node priorities*

As to the node priority bl_{in}, the schedule result is shown in Figure 5.8(c) with the schedule length of 18. In comparison, the classic dynamic list scheduling heuristics give scheule lengths of 23, 21 and 18 in Figure 4.5. Therefore, our advanced dynamic list scheduling heuristics can still get a better schedule result.

5.5.2 Comparison with Randomly Generated DAGs

Random DAGs have been used to evaluate the performance of the classic list scheduling heuristics in Section 4.4.2. We use the same parameters to generate random DAGs in this section, and the architecture is also the one given in Figure 5.5(b). The advanced list scheduling heuristics are compared to the classic static list scheduling heuristic with nodes sorted by their bottom levels to get accelerator factors (Acc).

Table 5.3: *Different dynamic node lists*

Node priorities	Dynamic node list
bl_{comp}	$n_1, n_4, n_2, n_6, n_7, n_3, n_8, n_9, n_5$
bl	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$
bl_{in}	$n_1, n_2, n_4, n_3, n_8, n_6, n_7, n_9, n_5$
bl_{out}	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$
bl_{io}	$n_1, n_2, n_4, n_3, n_8, n_7, n_6, n_9, n_5$

Advanced Static List Scheduling

Figure 5.9 gives the average Acc of the advanced static list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 0.1$. The average Acc of the combined static heuristic is also given in this figure. All the static heuristics have almost the same average Acc with $Acc \approx 1$. Since the techniques of critical child and communication delay are all aimed at optimizing the scheduling with communication contention, the improvement for the average Acc of the advanced static heuristics are almost not improved when the communication cost is very small in comparison with the computation cost. In fact, it is difficult to obtain different node lists when CCR is very small. Therefore, the schedule results for different node priorities are usually same, and the improvement for the average Acc of the combined advanced static heuristic is negligible.

Figure 5.10 gives the average Acc of the advanced static list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 1$. All the advanced static heuristics have $Acc > 1$, which means the advanced techniques are efficient to improve the schedule performance. Since different node lists can be obtained with different node priorities when the communication cost increases, the combined advanced static heuristic gives a much greater Acc. We also see that the advanced static heuristic with node priority of bl_{comp} & tl_{comp} usually gives the greatest Acc among all the five node priorities.

Figure 5.11 gives the average Acc of the advanced static list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 10$. Differing from that for $CCR = 1$, the advanced static heuristic with node priority of bl_{comp} & tl_{comp} gives the smallest average Acc among all the five groups of node priorities because the communication contention can not be neglected in this case. Since the five groups of node priorities usually give different node lists, the combined advanced static list scheduling heuristic improves greatly the final result.

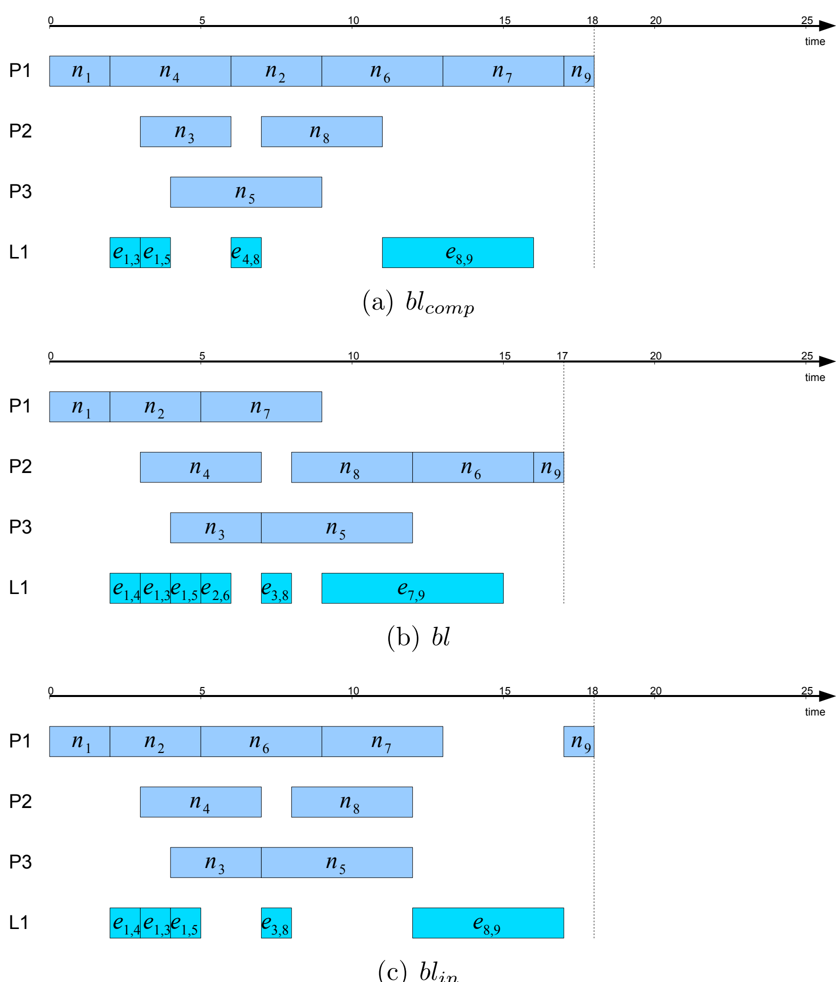


Figure 5.8: *Schedule results of advanced dynamic heuristic with different node priorities*

Figure 5.12 compares the average Acc of the combined advanced static list scheduling heuristic for different values of CCR. We can see that the average Acc increases when CCR varies from 0.1 to 10. If the number of nodes is fixed, the average Acc increases as the average in/out degree increases when $CCR = 10$. In fact, the critical child technique works well for scheduling nodes with many input edges especially when weights of edges are great; therefore, the Acc increases greatly as the average in/out degree increases when $CCR = 10$, but it is relatively stable when $CCR = 0.1$ and $CCR = 1$.

Figure 5.13 gives a comparison of the combined advanced static list scheduling heuristic to the combined classic static list scheduling heuristic. They have a similar average Acc for $CCR = 0.1$; however, the combined advanced static heuristic gives greater performance improvement on average in comparison with the combined classic

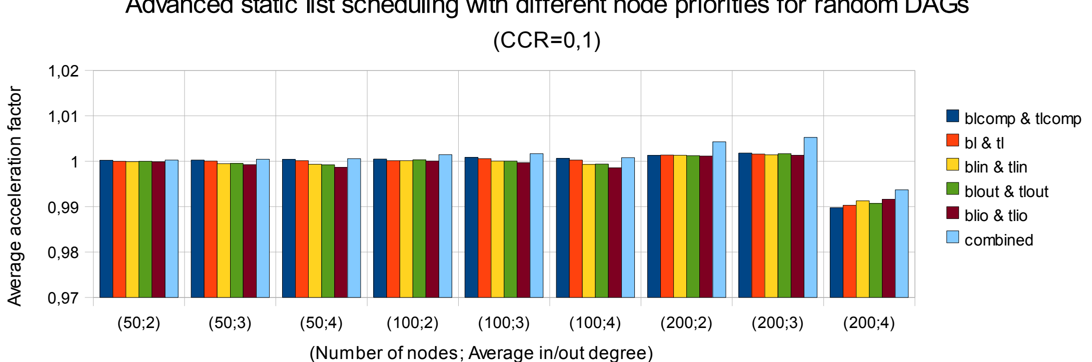


Figure 5.9: *Average Acc of advanced static heuristic ($CCR = 0.1$)*

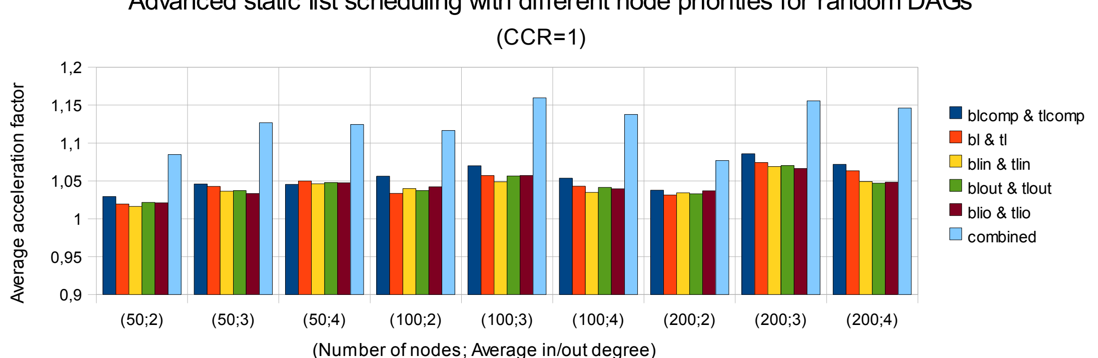


Figure 5.10: *Average Acc of advanced static heuristic ($CCR = 1$)*

static heuristic for both $CCR = 1$ and $CCR = 10$.

Advanced Dynamic List Scheduling

Figure 5.14 gives the average Acc of the advanced dynamic list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 0.1$. The average Acc decreases as the size of DAG increases. We do not get improvements from the advanced dynamic heuristics on average though they may give a good result for a specific DAG.

Figure 5.15 gives the average Acc of the advanced dynamic list scheduling heuristic with the five groups of node priorities for different groups of random DAGs when $CCR = 1$. Though a dynamic heuristic can not give improvements for all the groups of DAGs, the combined dynamic heuristic always gives $Acc > 1$, which means the schedule results are improved on average.

Figure 5.16 gives the average Acc of the advanced dynamic list scheduling heuristic

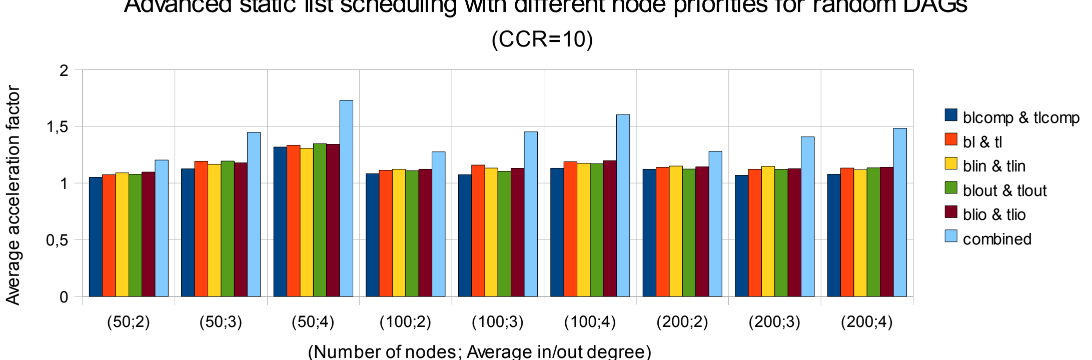


Figure 5.11: *Average Acc of advanced static heuristic ($CCR = 10$)*

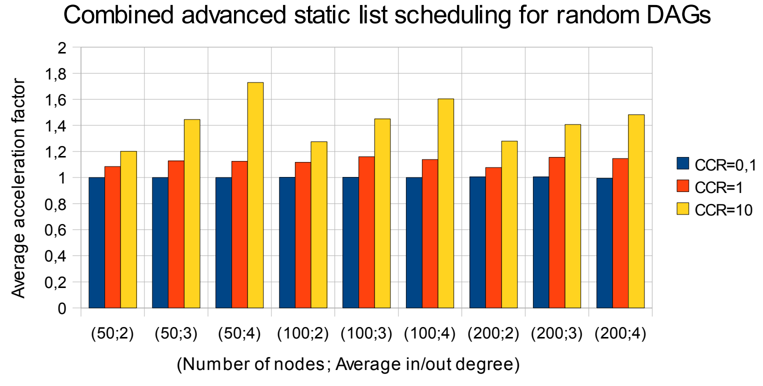


Figure 5.12: *Average Acc of combined advanced static heuristic*

with the five groups of node priorities for different groups of random DAGs when $CCR = 10$. Except for the heuristic with node priority of bl_{comp}, all the other four heuristics give $Acc > 1$ for each group of DAGs. Since the communication cost is much greater than the computation cost, the node priority without considering the communication becomes unpractical and usually gives bad results. However, the other four node priorities with communication give similar performances, and the combination of the different advanced dynamic heuristics improves greatly the final result.

Figure 5.17 compares the average Acc of the combined advanced dynamic list scheduling heuristic for different values of CCR. We can see that the average Acc increases when CCR varies from 0.1 to 10 for each group of DAGs. When the number of nodes is fixed, the Acc is relatively stable in the cases of $CCR = 0.1$ and $CCR = 1$, but it increases as the average in/out degree increases in the case of $CCR = 10$. This phenomenon is similar to that of the combined advanced static heuristic for the same

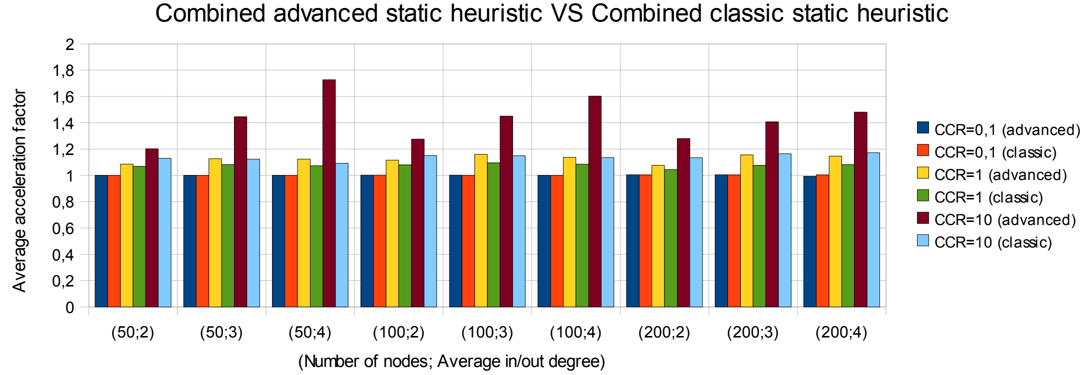


Figure 5.13: *Comparison of combined advanced static heuristic to combined classic static heuristic*

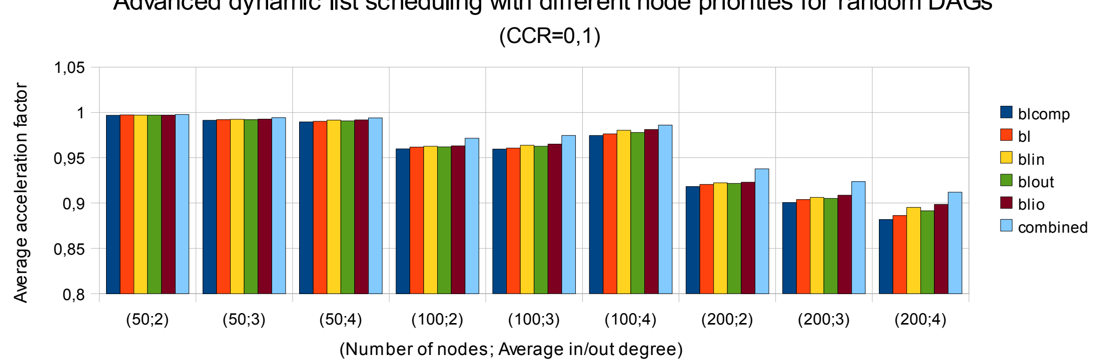


Figure 5.14: *Average Acc of advanced dynamic heuristic* ($CCR = 0.1$)

reason.

Figure 5.18 gives a comparison of the combined advanced dynamic list scheduling heuristic to the combined classic dynamic list scheduling heuristic. The combined advanced dynamic heuristic usually gives greater average Acc than the combined classic dynamic heuristic, especially in the cases of $CCR = 10$.

Static VS Dynamic

Figure 5.19 gives a comparison of the combined advanced static list scheduling heuristic to the combined advanced dynamic list scheduling heuristic. The combined static heuristic has a greater Acc when $CCR = 0.1$ and $CCR = 1$, but the combined dynamic heuristic usually gives better result than the combined static heuristic when $CCR = 10$. Therefore, the best way should be using all the combined heuristics (classic static, classic dynamic, advanced static, advanced dynamic) for a specific

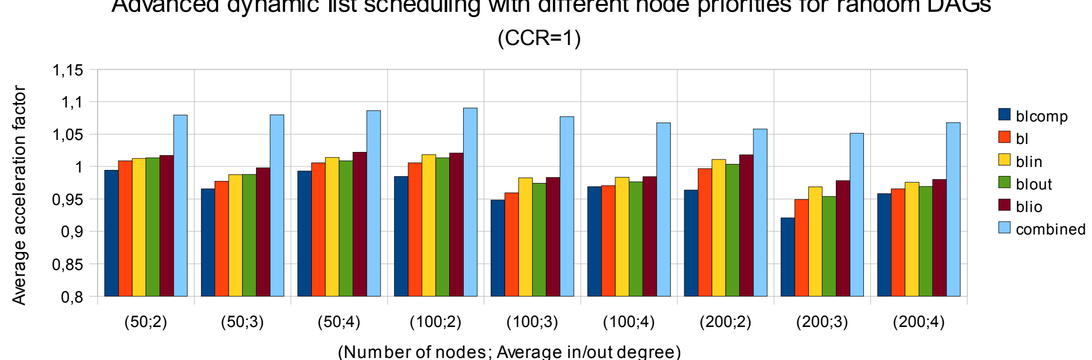


Figure 5.15: *Average Acc of advanced dynamic heuristic ($CCR = 1$)*

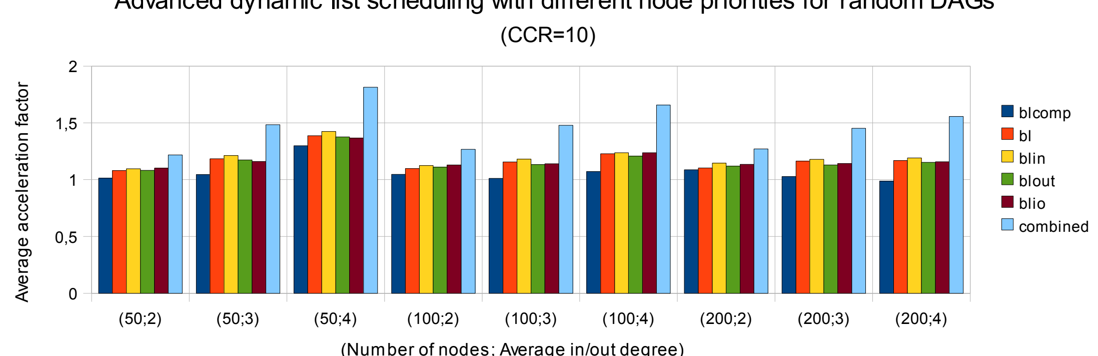


Figure 5.16: *Average Acc of advanced dynamic heuristic ($CCR = 10$)*

DAG and choosing the best schedule result.

5.6 Time Complexity of Advanced List Scheduling Heuristics

The time complexities of our advanced list scheduling heuristics are briefly presented as follows:

As analyzed in Section 4.5, the time complexity of the procedure `Schedule_Edge()` is $O\left(EO\left(routing\right)\right)$. Though the procedure `Schedule_Node()` additionally needs $O\left(\frac{E}{V}\right)$ time to calculate the ALAP, the total complexity of this procedure is still $O\left(\frac{E^2 O(routing)}{V} + \frac{V}{P}\right)$ on average.

As to the procedure `Select_Processor()`, it takes firstly $O\left(V\right)$ time to find the critical child $cc\left(n_i\right)$. When $cc\left(n_i\right)$ is found, given a specific processor p, it needs at

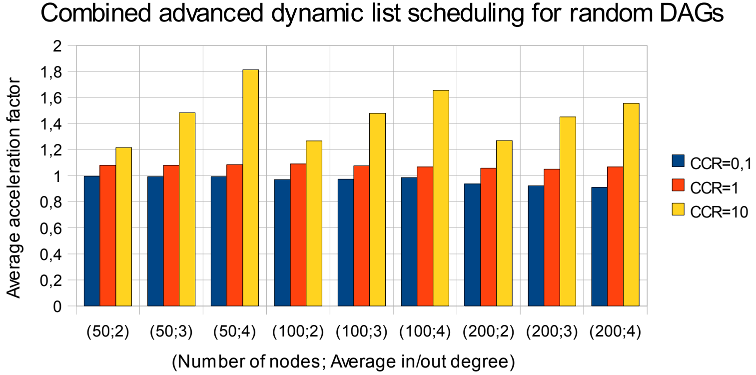


Figure 5.17: *Average Acc of combined advanced dynamic heuristic*

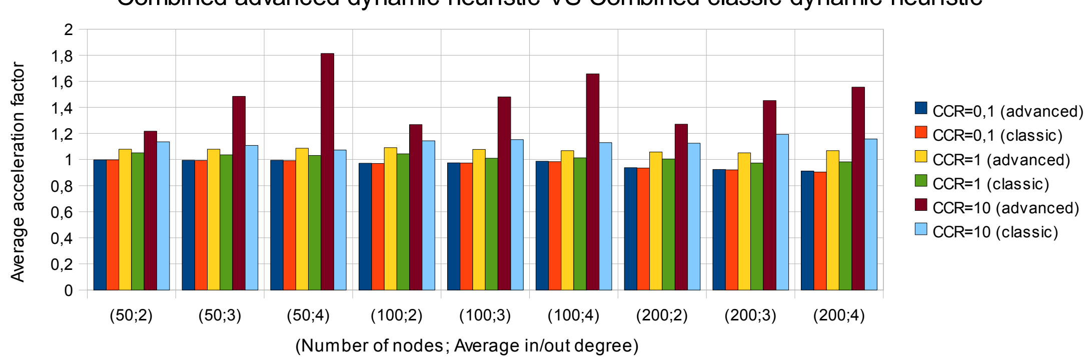


Figure 5.18: *Comparison of combined advanced dynamic heuristic to combined classic dynamic heuristic*

most $O(P)$ times of `Schedule_Node()` for the scheduling of n_i and $cc(n_i)$. Hence, the complexity in the outer for-loop is $O\left(P\left(\frac{E^2 O(routing)}{V}+\frac{V}{P}\right)\right)$, and the total complexity of `Select_Processor()` is $O\left(P\left(\frac{PE^2 O(routing)}{V}+V\right)\right)$.

In Algorithm 5.4, the time complexity lies on the procedure `Select_Processor()`. Therefore, the total complexity of the advanced static list scheduling heuristic is $O(P(PE^2 O(routing)+V^2))$. Similarly, the total complexity of the advanced dynamic list scheduling heuristic is also $O(P(PE^2 O(routing)+V^2))$.

Table 5.4 gives all the time complexities of the advanced list scheduling heuristics and the classic list scheduling heuristics with communication contention. The time complexities of our advanced list scheduling heuristics are only P times as those of the classic one given in Chapter 4.

Similar to the classic list scheduling heuristics, the advanced list scheduling heuristics can be combined with different node priorities, and we choose the best result for a

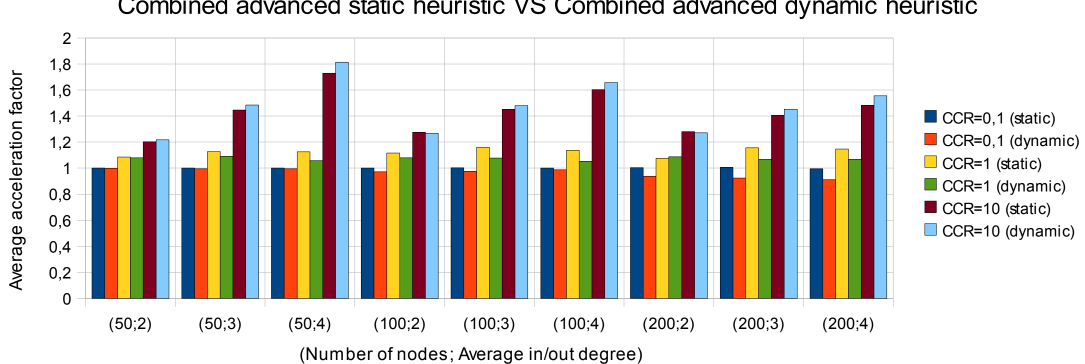


Figure 5.19: *Comparison of combined advanced static heuristic to combined advanced dynamic heuristic*

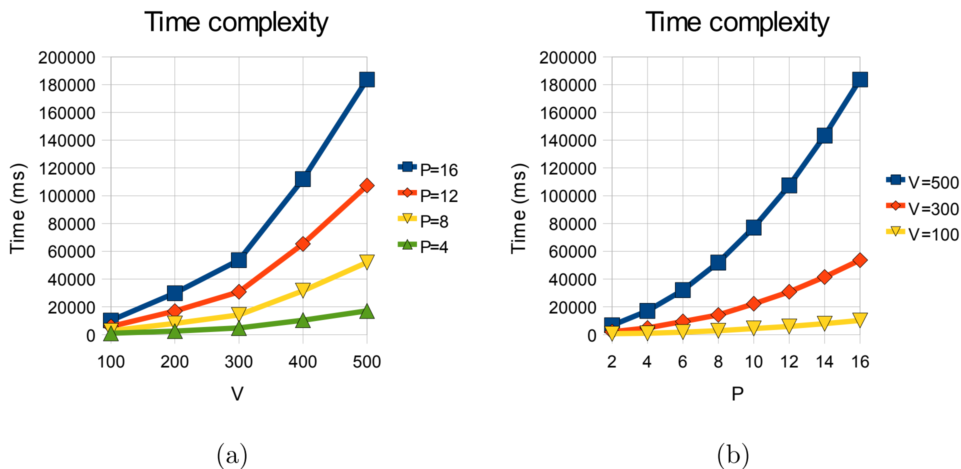
Table 5.4: *Time complexities of different list scheduling heuristics*

List scheduling heuristics	Time complexity
Classic static	$O\left(PE^2O\left(routing\right)+V^2\right)$
Classic dynamic	$O\left(PE^2O\left(routing\right)+V^2\right)$
Advanced static	$O\left(P\left(PE^2O\left(routing\right)+V^2\right)\right)$
Advanced dynamic	$O\left(P\left(PE^2O\left(routing\right)+V^2\right)\right)$

specific DAG. A combined advanced list scheduling heuristic consists of five advanced list scheduling heuristics with different node priorities. However, its complexity is unchanged and is always $O\left(P\left(PE^2O\left(routing\right)+V^2\right)\right)$.

Figure 5.20 shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined advanced static heuristic. All the DAGs have the average in/out degree of 4, and all the processors are connected to a switch. As shown in Figure 5.20(a) and Figure 5.20(b), the time increases with the square of V and also with the square of P. We run our heuristic on a Pentium Dual-Core PC at 2.4GHz, and it takes about 3 minutes to schedule a DAG with 500 nodes on an architecture of 16 processors. In fact, a complicated embedded application usually has less than 500 nodes in models of coarse and medium grain, and P is usually much smaller than V and E in a parallel embedded system. Therefore, the increase of time complexity is reasonable and acceptable for rapid prototyping.

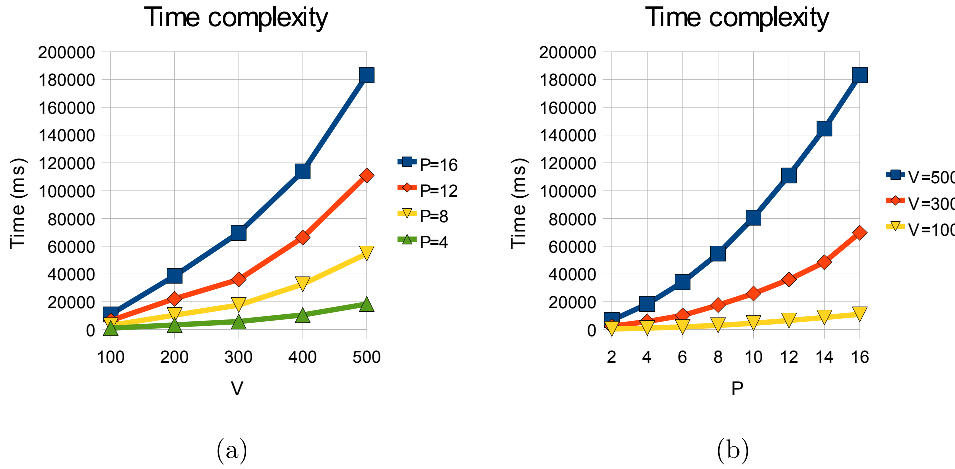
Figure 5.21 shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined advanced dynamic heuristic. The result is similar to that of the combined advanced static heuristic. The time increases with the square of V (Figure 5.21(a)) and also with the square of P



(a) (b)

Figure 5.20: *Time complexity of advanced static heuristic*

(Figure 5.21(b)). It takes about 3 minutes to schedule a DAG with 500 nodes on an architecture of 16 processors.



(a) (b)

Figure 5.21: *Time complexity of advanced dynamic heuristic*

A complicated embedded application usually has less than 500 nodes in models of coarse and medium grain, and P is usually much smaller than V and E in a parallel embedded system. Therefore, the increase of time complexity is reasonable and acceptable for rapid prototyping.

5.7 Conclusion

This chapter presented two advanced techniques for list scheduling with communication contention. The critical child technique helps to select a better processor for a node, and the communication delay technique enlarges idle time intervals on links.

The advanced techniques are combined with different node priorities to get combined advanced heuristics. Though the performance of the combined advanced static list scheduling heuristic is similar to that of the combined classic static list scheduling heuristic when $CCR = 0.1$, the combined advanced static heuristic gives greater performance improvement on average in comparison with the combined classic static heuristic when $CCR = 1$ and $CCR = 10$. As to the combined advanced dynamic list scheduling, the performance is almost not improved in the case of $CCR = 0.1$, but it is obviously improved in the cases of $CCR = 1$ and $CCR = 10$. The improvement increases as the communication to computation ratio increases for both the advanced static and dynamic heuristics. Since the communication cost is increasing from medium to high in modern digital communication and video compression applications, our method will work well for scheduling these applications on parallel embedded systems.

Since it is not certain which heuristic gives the best schedule result for a specific DAG, it will be reasonable to use all the combined heuristics (classic static, classic dynamic, advanced static, advanced dynamic) for a specific DAG and choosing the best result. Though the time complexity of the advanced heuristic is increased by a factor of P (the number of processors) in contrast to the classic one, it is acceptable because P is usually much smaller than V and E in a parallel embedded system. Therefore, our proposed method is reasonable and effective for rapid prototyping of complicated applications on parallel embedded systems.

Conclusions and Prospects

Conclusions

This work aimed at the prototyping methodology for parallel embedded systems. We first presented the methods of rapid prototyping and hardware/software co-design. Rapid prototyping is an important methodology for designing multiprocessor systems. As an example for rapid prototyping, SynDEx and SynDEx-Ic support the AAA rapid prototyping methodology. We used SynDEx for designing multi-MicroBlaze systems on FPGA. Hardware/software co-design is the tendency for designing modern embedded systems. Since hardware/software co-design needs to generate HDL code for a hardware coprocessor that is usually more complicated than software code, we presented the GAUT and OpenDF tools for generating HDL code from high-level languages. When considering an embedded system with multiple processors like MPSoC, the combination of rapid prototyping with hardware/software co-design becomes a good solution. We presented a new rapid prototyping framework of PREESM that supports hardware/software co-design for parallel embedded systems. PREESM firstly models an algorithm and an architecture as graphs, then it schedules the algorithm onto the architecture. The schedule results are finally used to generate code.

We mostly considered the scheduling problem in this work. As a first step for scheduling, algorithms and architectures were both modeled as graphs. We presented different graph models for algorithms and made a choice among them. The DAG

model was chosen to describe an algorithm because it was simple and described most information for task scheduling. We proposed an advanced architecture model to describe parallel embedded systems with distributed memory architecture. This model uses five kinds of vertices (processor, IP coprocessor, memory, communicator, and communication node) and two kinds of edges (bus and FIFO) to model different components of an embedded system. In addition, four functions are used to describe properties of the components. In comparison with the existing models such as the completely connected model, our advanced architecture model describes a parallel embedded system more accurately. This advanced architecture model was used for the task scheduling in this work.

Here task scheduling for parallel embedded systems is considered to be static, which means that the scheduling is done at compile time. After introducing the general task scheduling problem and giving a survey of the commonly used techniques for task scheduling, we formulated the task scheduling with our advanced architecture model. It consists in assigning computations and communications to components and finding time intervals on these components for the computations and communications. The aim is to get a minimum schedule length. A computation is executed on an operator that is a processor or an IP coprocessor. A communication is transferred on a route from one operator to another. A route usually contains several steps. Communications are handled in the cut-through mode on a route step and in the store-and-forward mode on different route steps. We defined the start and finish times of computations and communications on different components and gave the causality conditions with the advanced architecture model. Based on the causality conditions, the scheduling conditions were finally given to be fulfilled during the scheduling.

We simplified the advanced architecture model with a topology graph. The task scheduling problem is also simplified with the topology graph model and is indeed the task scheduling with communication contention. We researched advanced heuristics and techniques for this simplified task scheduling. In addition to the two existing groups of node levels for a DAG, we proposed three new groups of node levels by taking into account the communication contention. All these node levels have similar properties and can be used to sort nodes in topological order. We presented the classic list scheduling heuristic that can be classified as static or dynamic according to whether the node list is generated statically before the scheduling or dynamically during the scheduling. We also proposed two advanced techniques: the critical child technique helps to select a better processor for a node; the communication delay

technique enlarges idle time intervals on links. These two techniques are used to build up advanced static and dynamic list scheduling heuristics.

Using the five groups of node levels as priorities for the static and dynamic list scheduling usually leads to different scheduling orders of nodes and finally can give different scheduling results. We combined a list scheduling heuristic with the five groups of node levels and chose the best schedule result. Then we got four combined heuristics: classic static, classic dynamic, advanced static and advanced dynamic. We chose the classic static list scheduling with the node priority of bottom level as a standard for comparison. We used randomly generated DAGs to these heuristics in order to get statistical results. Experiments showed that all the combined heuristics got improvements in the case of medium or high communication. The improvements increase as the communication cost increases.

When comparing the combined static heuristics to the combined dynamic heuristics, their performances are similar. We also compared the combined advanced heuristics to the combined classic heuristics. Though the performances of the combined advanced heuristics are similar to those of the combined classic heuristics in the case of low communication, the combined advanced heuristics give greater improvements in the case of medium or high communication. An application can be accelerated up to 80%. The communication cost is increasing from medium to high in modern digital communication and video compression applications. Therefore, our methods will work well for scheduling these applications on parallel embedded systems.

Since it is not certain which heuristic gives the best schedule result for a specific DAG, it will be reasonable to use all the four combined heuristics for a specific DAG and choose the best result. Though the time complexities of the advanced heuristics are increased by a factor of P (the number of processors) in contrast to the classic ones, it is acceptable because P is usually much smaller than V (the number of nodes) and E (the number of edges) for a parallel embedded system. Therefore, our proposed method is reasonable and effective for rapid prototyping of complicated applications on parallel embedded systems.

Prospects

Our advanced techniques have been proven to be effective for the simplified task scheduling. The next step is to directly apply them for the task scheduling with the advanced architecture model. We also need to integrate our advanced heuristics into

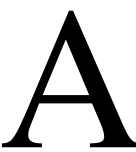
PREESM to cooperate with other tools. In fact, the schedule result of an application on a parallel embedded system should be further used to generate the code and finally to efficiently implement the application on the parallel embedded system. Since the advanced architecture model is closer to the real parallel embedded system than the simplified model, it will be straighter and easier to generate the code with the schedule result based on the advanced architecture model. We need to test it with some real applications like MPEG-4 video codec.

Since the task scheduling problem is NP-hard, our heuristics can just give some near-optimal results to shorten the time-to-market. Sometimes we may also need to optimize an application without the stress of the time-to-market. In this case, search-based methods like simulated annealing or genetic algorithms are the only way to get it. Therefore, finding search-based methods for the task scheduling with our advanced architecture model is also a research point as in [PMAN09].

Though the DAG model is simple to describe algorithms for task scheduling, it should be noticed that transforming a dataflow graph to a DAG may lose some information about the data between operations. For example, data can be broadcasted from one operation to several operations by a broadcasting edge in a dataflow graph. Such a broadcasting edge is usually transformed to several independent edges in a DAG because an edge must have exactly one origin node and one destination node. Therefore, it will be necessary to extend the DAG model by considering this kind of information in order to improve the final schedule result. In addition, transforming a dataflow model to a DAG model is not always straightforward. As a dataflow programming language, CAL is used to describe some applications like RVC. Transforming a CAL application to a DAG will be necessary for scheduling it on a parallel embedded system.

As to the future architecture of MPSoC, NoC is an emerging solution to connect different components on a chip [EVA06, DEL07]. A NoC can be modeled as a communication node in our advanced architecture model for simplicity. However, a NoC is not as ideal as a communication node when considering the details. We will need a more accurate model to describe the NoC. Extending the architecture model to include the NoC will be a challenge in our future work.

As a final conclusion, considering the experimental results and the open prospects, we can say that our approach is full of interest in reference to the objective of a prototyping methodology for parallel embedded systems.



IP-XACT Code of Advanced Architecture Model

A.1 TI's C6474 DSP

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:design xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4">
   <spirit:vendor>ietr.org</spirit:vendor>
   <spirit:name>C6474</spirit:name>
   <spirit:library>preesm</spirit:library>
   <spirit:version>1.0</spirit:version>
   <spirit:componentInstances>
      <spirit:componentInstance>
         <spirit:instanceName>P1</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="C64x+"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             processor</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="setupTime">
             (C1:1)</spirit:configurableElementValue>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>P2</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="C64x+"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             processor</spirit:configurableElementValue>
```

```
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="setupTime">
             (C1:1)</spirit:configurableElementValue>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>P3</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="C64x+"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             processor</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="setupTime">
             (C1:1)</spirit:configurableElementValue>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>CN1</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="SCR"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             communicationNode</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>B1</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="Bus"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             bus</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="dataRate">
             2</spirit:configurableElementValue>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>B2</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="Bus"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             bus</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="dataRate">
             2</spirit:configurableElementValue>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>B3</spirit:instanceName>
```

```
      <spirit:componentRef spirit:library="preesm" spirit:name="Bus"
       spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="componentType">
          bus</spirit:configurableElementValue>
         <spirit:configurableElementValue spirit:referenceId="refinement"/>
         <spirit:configurableElementValue spirit:referenceId="dataRate">
          2</spirit:configurableElementValue>
      </spirit:configurableElementValues>
   </spirit:componentInstance>
   <spirit:componentInstance>
      <spirit:instanceName>B4</spirit:instanceName>
      <spirit:componentRef spirit:library="preesm" spirit:name="Bus"
       spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="componentType">
          bus</spirit:configurableElementValue>
         <spirit:configurableElementValue spirit:referenceId="refinement"/>
         <spirit:configurableElementValue spirit:referenceId="dataRate">
          2</spirit:configurableElementValue>
      </spirit:configurableElementValues>
   </spirit:componentInstance>
   <spirit:componentInstance>
      <spirit:instanceName>C1</spirit:instanceName>
      <spirit:componentRef spirit:library="preesm" spirit:name="EDMA3.0"
       spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="componentType">
          communicator</spirit:configurableElementValue>
         <spirit:configurableElementValue spirit:referenceId="refinement"/>
      </spirit:configurableElementValues>
   </spirit:componentInstance>
   <spirit:componentInstance>
      <spirit:instanceName>IP1</spirit:instanceName>
      <spirit:componentRef spirit:library="preesm" spirit:name="VCP2"
       spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="componentType">
          ipCoprocessor</spirit:configurableElementValue>
         <spirit:configurableElementValue spirit:referenceId="refinement"/>
      </spirit:configurableElementValues>
   </spirit:componentInstance>
   <spirit:componentInstance>
      <spirit:instanceName>B5</spirit:instanceName>
      <spirit:componentRef spirit:library="preesm" spirit:name="Bus"
       spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="componentType">
          bus</spirit:configurableElementValue>
         <spirit:configurableElementValue spirit:referenceId="refinement"/>
         <spirit:configurableElementValue spirit:referenceId="dataRate">
          2</spirit:configurableElementValue>
      </spirit:configurableElementValues>
```

```
        </spirit:componentInstance>
        <spirit:componentInstance>
            <spirit:instanceName>B6</spirit:instanceName>
            <spirit:componentRef spirit:library="preesm" spirit:name="Bus"
             spirit:vendor="" spirit:version=""/>
            <spirit:configurableElementValues>
                <spirit:configurableElementValue spirit:referenceId="componentType">
                 bus</spirit:configurableElementValue>
                <spirit:configurableElementValue spirit:referenceId="refinement"/>
                <spirit:configurableElementValue spirit:referenceId="dataRate">
                 2</spirit:configurableElementValue>
            </spirit:configurableElementValues>
        </spirit:componentInstance>
        <spirit:componentInstance>
            <spirit:instanceName>IP2</spirit:instanceName>
            <spirit:componentRef spirit:library="preesm" spirit:name="TCP2"
             spirit:vendor="" spirit:version=""/>
            <spirit:configurableElementValues>
                <spirit:configurableElementValue spirit:referenceId="componentType">
                 ipCoprocessor</spirit:configurableElementValue>
                <spirit:configurableElementValue spirit:referenceId="refinement"/>
            </spirit:configurableElementValues>
        </spirit:componentInstance>
    </spirit:componentInstances>
    <spirit:interconnections>
        <spirit:interconnection>
            <spirit:name/>
            <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="P2"/>
            <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B2"/>
        </spirit:interconnection>
        <spirit:interconnection>
            <spirit:name/>
            <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="P1"/>
            <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B1"/>
        </spirit:interconnection>
        <spirit:interconnection>
            <spirit:name/>
            <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="CN1"/>
            <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B1"/>
        </spirit:interconnection>
        <spirit:interconnection>
            <spirit:name/>
            <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="CN1"/>
            <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B2"/>
        </spirit:interconnection>
        <spirit:interconnection>
            <spirit:name/>
            <spirit:activeInterface spirit:busRef="io3" spirit:componentRef="CN1"/>
            <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B3"/>
        </spirit:interconnection>
        <spirit:interconnection>
            <spirit:name/>
            <spirit:activeInterface spirit:busRef="io4" spirit:componentRef="CN1"/>
```

```
      <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B4"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P1"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P2"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P3"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>configure</spirit:displayName>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="P1"/>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="C1"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>configure</spirit:displayName>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="P2"/>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="C1"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>configure</spirit:displayName>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="P3"/>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="C1"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="IP1"/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B3"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="P3"/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B4"/>
   </spirit:interconnection>
   <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="C1"/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B5"/>
   </spirit:interconnection>
```

```
      <spirit:interconnection>
         <spirit:name/>
         <spirit:activeInterface spirit:busRef="io5" spirit:componentRef="CN1"/>
         <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B5"/>
      </spirit:interconnection>
      <spirit:interconnection>
         <spirit:name/>
         <spirit:activeInterface spirit:busRef="io6" spirit:componentRef="CN1"/>
         <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B6"/>
      </spirit:interconnection>
      <spirit:interconnection>
         <spirit:name/>
         <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="IP2"/>
         <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B6"/>
      </spirit:interconnection>
      <spirit:interconnection>
         <spirit:name/>
         <spirit:displayName>access</spirit:displayName>
         <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
         <spirit:activeInterface spirit:busRef="a" spirit:componentRef="IP2"/>
      </spirit:interconnection>
      <spirit:interconnection>
         <spirit:name/>
         <spirit:displayName>access</spirit:displayName>
         <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
         <spirit:activeInterface spirit:busRef="a" spirit:componentRef="IP1"/>
      </spirit:interconnection>
   </spirit:interconnections>
   <spirit:hierConnections/>
</spirit:design>
```

A.2 Xilinx's FPGA-based MPSoC

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:design xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4">
   <spirit:vendor>ietr.org</spirit:vendor>
   <spirit:name>2P2IP</spirit:name>
   <spirit:library>preesm</spirit:library>
   <spirit:version>1.0</spirit:version>
   <spirit:componentInstances>
      <spirit:componentInstance>
         <spirit:instanceName>P1</spirit:instanceName>
         <spirit:componentRef spirit:library="preesm" spirit:name="microblaze"
          spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             processor</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="setupTime">
             (C1:10)</spirit:configurableElementValue>
```

```
        </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
        <spirit:instanceName>P2</spirit:instanceName>
        <spirit:componentRef spirit:library="preesm" spirit:name="microblaze"
         spirit:vendor="" spirit:version=""/>
        <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             processor</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
            <spirit:configurableElementValue spirit:referenceId="setupTime">
             (C2:10)</spirit:configurableElementValue>
        </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
        <spirit:instanceName>IP1</spirit:instanceName>
        <spirit:componentRef spirit:library="preesm" spirit:name="IP1"
         spirit:vendor="" spirit:version=""/>
        <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             ipCoprocessor</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
        </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
        <spirit:instanceName>IP2</spirit:instanceName>
        <spirit:componentRef spirit:library="preesm" spirit:name="IP2"
         spirit:vendor="" spirit:version=""/>
        <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             ipCoprocessor</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
        </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
        <spirit:instanceName>C1</spirit:instanceName>
        <spirit:componentRef spirit:library="preesm" spirit:name="dma"
         spirit:vendor="" spirit:version=""/>
        <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             communicator</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
        </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
        <spirit:instanceName>C2</spirit:instanceName>
        <spirit:componentRef spirit:library="preesm" spirit:name="dma"
         spirit:vendor="" spirit:version=""/>
        <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="componentType">
             communicator</spirit:configurableElementValue>
            <spirit:configurableElementValue spirit:referenceId="refinement"/>
        </spirit:configurableElementValues>
```

```
    </spirit:componentInstance>
    <spirit:componentInstance>
       <spirit:instanceName>CN1</spirit:instanceName>
       <spirit:componentRef spirit:library="preesm" spirit:name="crossbar"
        spirit:vendor="" spirit:version=""/>
       <spirit:configurableElementValues>
          <spirit:configurableElementValue spirit:referenceId="componentType">
           communicationNode</spirit:configurableElementValue>
          <spirit:configurableElementValue spirit:referenceId="refinement"/>
       </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
       <spirit:instanceName>B1</spirit:instanceName>
       <spirit:componentRef spirit:library="preesm" spirit:name="plb"
        spirit:vendor="" spirit:version=""/>
       <spirit:configurableElementValues>
          <spirit:configurableElementValue spirit:referenceId="componentType">
           bus</spirit:configurableElementValue>
          <spirit:configurableElementValue spirit:referenceId="refinement"/>
          <spirit:configurableElementValue spirit:referenceId="dataRate">
           2.0</spirit:configurableElementValue>
       </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
       <spirit:instanceName>B2</spirit:instanceName>
       <spirit:componentRef spirit:library="preesm" spirit:name="plb"
        spirit:vendor="" spirit:version=""/>
       <spirit:configurableElementValues>
          <spirit:configurableElementValue spirit:referenceId="componentType">
           bus</spirit:configurableElementValue>
          <spirit:configurableElementValue spirit:referenceId="refinement"/>
          <spirit:configurableElementValue spirit:referenceId="dataRate">
           2.0</spirit:configurableElementValue>
       </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
       <spirit:instanceName>F1</spirit:instanceName>
       <spirit:componentRef spirit:library="preesm" spirit:name="fsl"
        spirit:vendor="" spirit:version=""/>
       <spirit:configurableElementValues>
          <spirit:configurableElementValue spirit:referenceId="componentType">
           fifo</spirit:configurableElementValue>
          <spirit:configurableElementValue spirit:referenceId="refinement"/>
          <spirit:configurableElementValue spirit:referenceId="dataRate">
           0.5</spirit:configurableElementValue>
       </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
       <spirit:instanceName>F2</spirit:instanceName>
       <spirit:componentRef spirit:library="preesm" spirit:name="fsl"
        spirit:vendor="" spirit:version=""/>
       <spirit:configurableElementValues>
          <spirit:configurableElementValue spirit:referenceId="componentType">
```

```
          fifo</spirit:configurableElementValue>
          <spirit:configurableElementValue spirit:referenceId="refinement"/>
          <spirit:configurableElementValue spirit:referenceId="dataRate">
           0.5</spirit:configurableElementValue>
      </spirit:configurableElementValues>
    </spirit:componentInstance>
  </spirit:componentInstances>
  <spirit:interconnections>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="P1"/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="C1"/>
      <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="CN1"/>
      <spirit:activeInterface spirit:busRef="io3" spirit:componentRef="B1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="P2"/>
      <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="B2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="C2"/>
      <spirit:activeInterface spirit:busRef="io3" spirit:componentRef="B2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io2" spirit:componentRef="CN1"/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="B2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:activeInterface spirit:busRef="io1" spirit:componentRef="IP2"/>
      <spirit:activeInterface spirit:busRef="io4" spirit:componentRef="B2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>directed</spirit:displayName>
      <spirit:activeInterface spirit:busRef="o1" spirit:componentRef="P1"/>
      <spirit:activeInterface spirit:busRef="i1" spirit:componentRef="F1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>directed</spirit:displayName>
      <spirit:activeInterface spirit:busRef="o1" spirit:componentRef="F1"/>
```

```
      <spirit:activeInterface spirit:busRef="i1" spirit:componentRef="IP1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>directed</spirit:displayName>
      <spirit:activeInterface spirit:busRef="o1" spirit:componentRef="IP1"/>
      <spirit:activeInterface spirit:busRef="i1" spirit:componentRef="F2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>directed</spirit:displayName>
      <spirit:activeInterface spirit:busRef="o1" spirit:componentRef="F2"/>
      <spirit:activeInterface spirit:busRef="i1" spirit:componentRef="P1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P1"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="IP1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C2"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="IP2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C2"/>
      <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>configure</spirit:displayName>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="P1"/>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="C1"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>configure</spirit:displayName>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="P2"/>
      <spirit:activeInterface spirit:busRef="c" spirit:componentRef="C2"/>
    </spirit:interconnection>
    <spirit:interconnection>
      <spirit:name/>
      <spirit:displayName>access</spirit:displayName>
```

```
        <spirit:activeInterface spirit:busRef="a" spirit:componentRef="C1"/>
        <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P1"/>
      </spirit:interconnection>
      <spirit:interconnection>
        <spirit:name/>
        <spirit:displayName>access</spirit:displayName>
        <spirit:activeInterface spirit:busRef="a" spirit:componentRef="P2"/>
        <spirit:activeInterface spirit:busRef="a" spirit:componentRef="IP2"/>
      </spirit:interconnection>
    </spirit:interconnections>
    <spirit:hierConnections/>
</spirit:design>
```

List of Figures

1.1 SynDEx design flow 7
1.2 Rapid prototyping design flow with SynDEx 9
1.3 SynDEx kernel organization 10
1.4 SynDEx-Ic design flow 13
1.5 GAUT design flow 14
1.6 GAUT tool structure 15
1.7 A simple CAL network 18
1.8 CAL2HDL tool structure in OpenDF 19
1.9 Algorithm of MPEG-4 Part 2 decoder 20
1.10 Architecture of PC+ML402 21
1.11 An Eclipse-based rapid prototyping framework 23
1.12 Input/output with Graphiti's XML format $\mathcal{G}$ 24
1.13 A sample graph 25
1.14 The type of vertices of the graph shown in Figure 1.13 26
1.15 The type of edges of the graph shown in Figure 1.13 26
1.16 A workflow graph: From SDF and IP-XACT descriptions to code . . 27

2.1 (a) An SDF actor; (b) An SDF graph 32
2.2 Control actors: SWITCH and SELECT 33
2.3 Containing relation of different dataflow models 34

2.4 A DAG example . . . 37
2.5 A topological order of the DAG in Figure 2.4 . . . 37
2.6 Top and bottom levels . . . 39
2.7 Node dependency of the recursive definition of node levels . . . 39
2.8 Shared memory architecture . . . 44
2.9 Distributed memory architecture . . . 44
2.10 Examples of static networks . . . 45
2.11 Legend of vertices and edges . . . 47
2.12 Organization of the architecture model . . . 48
2.13 Multicore DSP architecture of Texas Instruments . . . 50
2.14 Multicore DSP architecture of Texas Instruments . . . 51
2.15 A multi-DSP architecture . . . 51
2.16 MPSoC architecture on the FPGA of Xilinx . . . 52
2.17 Component instance of a FIFO . . . 53
2.18 Interconnection from a processor to a FIFO . . . 53

3.1 Scheduling without/with communication cost . . . 57
3.2 Linear clustering . . . 59
3.3 Scheduling with node duplication . . . 61
3.4 Scheduling on an architecture with shared bus . . . 62
3.5 Two routing modes . . . 64
3.6 Performing communication by communicator . . . 67
3.7 Performing communication by processor . . . 67
3.8 Comparison of two models . . . 72
3.9 Examples of architecture . . . 73

4.1 Three new groups of node levels . . . 78
4.2 A DAG example . . . 79
4.3 Examples of architecture . . . 87
4.4 Schedule results of static heuristic with different node priorities . . . 88
4.5 Schedule results of dynamic heuristic with different node priorities . . . 90
4.6 Average Acc of static heuristic ($CCR = 0.1$) . . . 91
4.7 Average Acc of static heuristic ($CCR = 1$) . . . 92
4.8 Average Acc of static heuristic ($CCR = 10$) . . . 92
4.9 Average Acc of combined static heuristic . . . 93
4.10 Average Acc of dynamic heuristic ($CCR = 0.1$) . . . 93

4.11 Average Acc of dynamic heuristic ($CCR = 1$) 94
4.12 Average Acc of dynamic heuristic ($CCR = 10$) 94
4.13 Average Acc of combined dynamic heuristic 95
4.14 Comparison of combined static heuristic to combined dynamic heuristic 95
4.15 Time complexity of static heuristic 97
4.16 Time complexity of dynamic heuristic 97

5.1 A join DAG and two different schedule results 100
5.2 Communication delay 102
5.3 A DAG example 104
5.4 Scheduling procedures with/without communication delay 104
5.5 Examples of architecture 106
5.6 A DAG example 106
5.7 Schedule results of advanced static heuristic with different node priorities 108
5.8 Schedule results of advanced dynamic heuristic with different node priorities 110
5.9 Average Acc of advanced static heuristic ($CCR = 0.1$) 111
5.10 Average Acc of advanced static heuristic ($CCR = 1$) 111
5.11 Average Acc of advanced static heuristic ($CCR = 10$) 112
5.12 Average Acc of combined advanced static heuristic 112
5.13 Comparison of combined advanced static heuristic to combined classic static heuristic 113
5.14 Average Acc of advanced dynamic heuristic ($CCR = 0.1$) 113
5.15 Average Acc of advanced dynamic heuristic ($CCR = 1$) 114
5.16 Average Acc of advanced dynamic heuristic ($CCR = 10$) 114
5.17 Average Acc of combined advanced dynamic heuristic 115
5.18 Comparison of combined advanced dynamic heuristic to combined classic dynamic heuristic 115
5.19 Comparison of combined advanced static heuristic to combined advanced dynamic heuristic 116
5.20 Time complexity of advanced static heuristic 117
5.21 Time complexity of advanced dynamic heuristic 117

List of Tables

2.1 Different node levels for the DAG in Figure 2.4 41
2.2 Different topological orders 42

4.1 Input top level and bottom level 79
4.2 Output top level and bottom level 80
4.3 Input/output top level and bottom level 81
4.4 Different topological orders 82
4.5 Different static node lists 87
4.6 Different dynamic node lists 89

5.1 Different static node lists 107
5.2 Critical children according to different node priorities 107
5.3 Different dynamic node lists 109
5.4 Time complexities of different list scheduling heuristics 116

List of Algorithms

2.1 **Topological_Sort**(G) 36

2.2 **DFS_Visit**(n_i,NL) 36

2.3 **Compute_Top_Level**(G) 40

2.4 **Compute_Bottom_Level**(G) 41

3.1 **General_Static_List_Scheduling**(G, P) 60

3.2 **General_Dynamic_List_Scheduling**(G, P) 60

4.1 **Static_List_Scheduling**(G, TG) 82

4.2 **Select_Processor**(n_i, P) 84

4.3 **Schedule_Node**(n_i, p) 84

4.4 **Schedule_Edge**(e_{ij}, p) 85

4.5 **Dynamic_List_Scheduling**(G, TG) 85

4.6 **Choose_Node**(UN) 86

5.1 **Select_Processor**(n_i, P) 101

5.2 **Schedule_Node**(n_i, p, $isTemporary$) 103

5.3 **Schedule_Edge**(e_{ij}, p) 103

5.4 **Advanced_Static_List_Scheduling**(G, TG) 105

5.5 **Advanced_Dynamic_List_Scheduling**(G, TG) 105

Personal Publications

[1] Pengcheng Mu, Michaël Raulet, Jean-François Nezan and Jean-Gabriel Cousin. Automatic Code Generation for Multi-MicroBlaze System with SynDEx. In 15th European Signal Processing Conference (EUSIPCO 2007), Poznan, Pologne, September 2007.

[2] Jonathan Piat, Mickaël Raulet, Maxime Pelcat, Pengcheng Mu and Olivier Déforges. An Extensible Framework for Fast Prototyping of Multiprocessor Dataflow Applications. In IDT'08: Proceedings of the 3rd International Design and Test Workshop, Monastir, Tunisia, December 2008.

[3] Pengcheng Mu, Jean-Gabriel Cousin, Jean-François Nezan and Mickaël Raulet. Heuristique statique améliorée d'ordonnancement de tâches : impact sur le tri des tâches et sur l'allocation de processeur. In XXIIe Colloque GRETSI, Dijon, France, September 2009.

[4] Pengcheng Mu, Jean-François Nezan, Michaël Raulet and Jean-Gabriel Cousin. A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention. In DASIP 2009: Conference on Design and Architectures for Signal and Image Processing, Sophia Antipolis, France, September 2009. (Submitted)

[5] Pengcheng Mu, Jean-François Nezan, Jean-Gabriel Cousin and Michaël Raulet. A Dynamic List Scheduling Heuristic with Communication Contention in Parallel Embedded Systems: New Node Priorities, Critical Child and Communica-

tion Delay. In EMSOFT 2009: International Conference on Embedded Software, Grenoble, France, October 2009. (Submitted)

Bibliography

[ACD74] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974. 56, 60

[AkK98] Ishfaq Ahmad and Yu kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9:872–892, 1998. 61

[BBR02] Olivier Beaumont, Vincent Boudet, and Yves Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 37, Washington, DC, USA, 2002. IEEE Computer Society. 62

[BEH+01] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. Graphml progress report, structural layer proposal. In P Mutzel, M Junger, and S Leipert, editors, *Graph Drawing - 9th International Symposium, GD 2001 Vienna Austria,*, pages 501–512, Heidelberg, 2001. Springer Verlag. 26

[BEP+07] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. *Handbook on Scheduling: From Theory to Applications.* Springer-Verlag, 2007. 55

[BK06] Peter Brucker and Sigrid Knust. *Complex Scheduling.* Springer-Verlag, 2006. 55

[BMB05] Pierre Bomel, Eric Martin, and Emmanuel Boutillon. Synchronization Processor Synthesis for Latency Insensitive Systems. In EDAA European design and Automation Association, editors, *Design, Automation and Test in Europe DATE'05*, volume 2, pages 896–897, Munich Allemagne, 03 2005. Submitted on behalf of EDAA (http://www.edaa.com/). 14

[Bru07] Peter Brucker. *Scheduling Algorithms.* Springer-Verlag, 5th edition, 2007. 55, 58

[Buc93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, EECS Department, University of California, Berkeley, 1993. 33

[CC91] J. Y. Colin and P. Chretienne. C.P.M. Scheduling with Small Communication Delays and Task Duplication. *OPERATIONS RESEARCH*, 39(4):680–684, 1991. 61

[CCB+05] Philippe Coussy, Gwenolé Corre, Pierre Bomel, Eric Senn, and Eric Martin. High-level synthesis under I/O Timing and Memory constraints. In IEEE, editor, *International Symposium on Circuits And Systems*, pages 680–683. IEEE, 2005. 14

[CDKM02] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in Real-Time Systems.* John Wiley & Sons Ltd, 2002. 55

[CJ01] B. Cirou and E. Jeannot. Triplet: A clustering scheduling algorithm for heterogeneous systems. In *Parallel Processing Workshops, 2001. International Conference on*, pages 231–236, 2001. 58

[CKP+93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993. 30

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001. 35

[CR92] Y.-C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on dis-

tributed memory multiprocessors. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 512–521, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 61

[CSB+04] Gwenolé Corre, Eric Senn, Pierre Bomel, Nathalie Julien, and Eric Martin. Memory Accesses management during High Level Synthesis. In ACM, editor, *IEEE ACM CO-DESign symposium and International Symposium on System Synthesis IEEE ACM CO-DESign symposium and International Symposium on System Synthesis*, pages 42–47, stockholm Sweden, 2004. SIGDA ACM. ISBN : 1-58113-937-3. 14

[DAYA02] Muhammad K. Dhodhi, Imtiaz Ahmad, Anwar Yatama, and Ishfaq Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 62(9):1338–1361, 2002. 61

[DEL07] Julien DELORME. *Méthodologie de modélisation et d'exploration d'architecture de réseaux sur puce appliquée aux télécommunications.* PhD thesis, INSA de Rennes, 2007. 122

[Den74] Jack B. Dennis. First version of a data flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 1974. 31

[Die05] Reinhard Diestel. *Graph Theory.* Springer-Verlag, 3rd edition, 2005. 29

[Dun90] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, Feb 1990. 29

[EJ03] Johan Eker and Jörn W. Janneck. CAL Language Report. Technical report, ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003. 17

[ERA94] Hesham El-Rewini and Hesham H. Ali. On considering communication in scheduling task graphs on parallel processors. *International Journal of Parallel, Emergent and Distributed Systems*, 3(3):177–191, 1994. 56

[ERAEB05] Hesham El-Rewini and Mostafa Adb-El-Barr. *Advanced Computer Architecture and Parallel Processing.* John Wiley & Sons, Inc., 2005. 29

[ERL90] Hesham El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.*, 9(2):138–153, 1990. 62

[ERLA94] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task scheduling in parallel and distributed systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. 58

[EVA06] Samuel EVAIN. *μSpider Environnement de Conception de Réseau sur Puce.* PhD thesis, INSA de Rennes, 2006. 122

[Fly66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966. 43

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM. 29, 43

[GG69] R. L. Graham and R. L. Grahamt. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969. 60

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979. 12, 58

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999. 6, 7

[GS03] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003. 30

[Gu93] J. Gu. Local search for satisfiability (sat) problem. *Systems, Man and Cybernetics, IEEE Transactions on*, 23(4):1108–1129, Jul/Aug 1993. 61

[GY93] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993. 58

[HCAL89] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989. 56, 60

[HH04] Randy L. Haupt and Sue Ellen Haupt. *Pratical Genetic Algorithms*. John Wiley & Sons, Inc., 2nd edition, 2004. 61

[HJ05] T. Hagras and J. Janeček. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Comput.*, 31(7):653–670, 2005. 61

[HM95] C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. In *Emerging Technologies and Factory Automation, 1995. ETFA '95, Proceedings., 1995 INRIA/IEEE Symposium on*, volume 1, pages 167–189 vol.1, Oct 1995. 58

[HP02] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 3 edition, June 2002. 64

[Hu61] T. C. Hu. Parallel Sequencing and Assembly Line Problems. *OPERATIONS RESEARCH*, 9(6):841–848, 1961. 59

[Jan07] Jörn W. Janneck. NL - a Network Language. Technical report, ASTG Technical Memo, Programmable Solutions Group, Xilinx Inc., July 2007. 17

[JE01] Jörn W. Janneck and Robert Esser. A predicate-based approach to defining visual language syntax. In *In Symposium on Visual Languages and Formal Methods, HCC01, Stresa*, pages 40–47, 2001. 24

[JMP+08] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs: an mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SIPS'08. (Best paper). IEEE Workshop on*, Washington D.C., USA, 2008. 16, 19, 21

[KA95] Yu-Kwong Kwok and I. Ahmad. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 36, Washington, DC, USA, 1995. IEEE Computer Society. 62

[KA96] Yu-Kwong Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs onto multiprocessors. *IEEE*

Transactions on Parallel and Distributed Systems, 7(5):506–521, May 1996. 58, 100, 101

[KA97] Yu-Kwong Kwok and Ishfaq Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *J. Parallel Distrib. Comput.*, 47(1):58–77, 1997. 61

[KA99a] Yu-Kwong Kwok and Ishfaq Ahmad. Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors. In *International Conference on Parallel Processing*, pages 551–558, 1999. 60, 62

[KA99b] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999. 35

[KAG96] Yu-Kwong Kwok, Ishfaq Amad, and Jun Gu. Fast: A low-complexity algorithm for efficient scheduling of dags on parallel processors. In *Proceedings of the 1996 Internationnal Conference on Parallel Processing*, pages 150–157, August 1996. 61

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam. 31

[KASG03] L. Kaouane, M. Akil, Y. Sorel, and T. Grandpierre. A methodology to implement real-time applications on reconfigurable circuits. In *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'03*, Las Vegas, USA, June 2003. 12

[KL88] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *Software, IEEE*, 5(1):23–32, Jan 1988. 58, 61

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Proccedings of the IFIP Congress, Toronto, Canada*, pages 993–998, 1977. 31

[KN84] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *Computers, IEEE Transactions on*, C-33(11):1023–1029, Nov. 1984. 60

[KS93] S. Kon'ya and T. Satoh. Task scheduling on a hypercube with link contentions. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 363–368, Apr 1993. 62

[LGCH+05] Bertrand Le Gal, Emmanuel Casseau, Sylvain Huet, Pierre Bomel, Christophe Jego, and Eric Martin. C-based rapid prototyping for digital signal processing. In *Proceedings of 13th European Signal Processing Conference*, Antalya, Turkey, 2005. 14

[LGCHM05] Bertrand Le Gal, Emmanuel Casseau, Sylvain Huet, and Eric Martin. Pipelined memory controllers for DSP applications handling unpredictable data acesses. In *IEEE Computer Society Annual Symposium on VLSI*, pages 268 – 269. IEEE, 2005. 14

[LM87a] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987. 32

[LM87b] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 25, 32

[LMTJ07] Christophe Lucarz, Marco Mattavelli, Joseph Thomas-Kerr, and Jorn Janneck. Reconfigurable media coding: A new specification model for multimedia coders. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 481–486, 2007. 17, 19

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. 32

[LP98] Jing-Chiou Liou and Michael A. Palis. A new heuristic for scheduling parallel programs on multiprocessor. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 358, Washington, DC, USA, 1998. IEEE Computer Society. 61

[LPX05] G. Q. Liu, K. L. Poh, and M. Xie. Iterative list scheduling for heterogeneous computing. *J. Parallel Distrib. Comput.*, 65(5):654–665, 2005. 62

[Mar06] Grant Martin. Overview of the mpsoc design challenge. In *Proceedings of the 43rd annual conference on Design automation*, San Francisco, CA, USA, July 2006. 5

[MG89] Carolyn McCreary and Helen Gill. Automatic determination of grain size for efficient parallel processing. *Commun. ACM*, 32(9):1073–1078, 1989. 58

[MG94] Neelima Mehdiratta and Kanad Ghose. A bottom-up approach to task scheduling on distributed memory multiprocessors. In *ICPP '94: Proceedings of the 1994 International Conference on Parallel Processing*, pages 151–154, Washington, DC, USA, 1994. IEEE Computer Society. 62

[MG97] G. De Micheli and R. K. Gupta. Hardware/software co-design. *Proceeding of the IEEE*, 85(3):349–365, March 1997. 5

[MSP+95] Daniel A. Menascé, Debanjan Saha, Stella C. da Silva Porto, Virgilio A. F. Almeida, and Satish K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *J. Parallel Distrib. Comput.*, 28(1):1–18, 1995. 62

[OH96] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, pages 573–577, London, UK, 1996. Springer-Verlag. 62

[OH06] Salim Ouadjaout and Dominique Houzet. Generation of embedded hardware/software from systemc. *EURASIP J. Embedded Syst.*, 2006(1):19–19, 2006. 5

[PLW96] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):46–55, 1996. 61

[PMAN09] Maxime Pelcat, Pierrick Menuet, Slaheddine Aridhi, and Jean-François Nezan. Scalable compile-time scheduler for multi-core architectures. In *DATE'09*, Nice, France, April 2009. 122

[PRP+08] Jonathan Piat, Mickaël Raulet, Maxime Pelcat, Pengcheng Mu, and Olivier Déforges. An extensible framework for fast prototyping of multiprocessor dataflow applications. In *IDT'08: Proceedings of the 3rd International Design and Test Workshop*, Monastir, Tunisia, december 2008. 27

[PY90] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19(2):322–328, 1990. 61

[RRND06] Ghislain Roquier, Michaël Raulet, Jean-François Nezan, and Olivier Déforges. Using RTOS in the AAA methodology automatic executive gen-

eration. In *Proceedings of 14th European Signal Processing Conference*, Florence, Italy, 2006. 10

[RS87] V. J. Rayward-Smith. Uet scheduling with unit interprocessor communication delays. *Discrete Appl. Math.*, 18(1):55–71, 1987. 56

[RUN+05a] M. Raulet, F. Urban, J.-F. Nezan, C. Moy, and O. Déforges. Syndex executive kernels for fast developments of applications over heterogeneous architectures. In *Proceedings of 13th European Signal Processing Conference*, Antalya, Turkey, 2005. 10

[RUN+05b] M. Raulet, F. Urban, J.-F. Nezan, C. Moy, O. Déforges, and Y. Sorel. Rapid Prototyping For Heterogeneous Multicomponent Systems: An MPEG-4 Stream Over An UMTS Communication Link. *Journal Of Applied Signal Processing (JASP)*, 2005. 6, 7, 8, 9

[RWR+08] Ghislain Roquier, Matthieu Wipliez, Mickaël Raulet, Jörn W. Janneck, Ian D. Miller, and David B. Parlour. Automatic software synthesis of dataflow program: An mpeg-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SIPS'08. IEEE Workshop on*, Washington, D.C., USA, 2008. 16, 21

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989. 55, 58

[SB00] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors - Scheduling and Synchronization*. Marcel Dekker, Inc., 2000. 55

[SBB06] Abdelhalim Samahi, El-Bay Bourennane, and Sami Boukhechem. Communication interface generation for hw/sw architecture in the starsoc environment. pages 1–6, Sept. 2006. 5

[SD08] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Springer-Verlag, 2008. 61

[SG91] Rok Sosic and Jun Gu. Fast search algorithms for the n-queens problem. *IEEE Trans. on Systems, Man, and Cybernetics*, 21:1572–1576, 1991. 61

[SGB06] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. 89

[Sin07] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, 2007. 55, 77

[SL93a] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4:175–187, Feb. 1993. 60, 62

[SL93b] G.C. Sih and E.A. Lee. Declustering: a new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):625–637, June 1993. 58

[SPI08] SPIRIT Schema Working Group. IP-XACT v1.4: A specification for XML meta-data and tool interfaces. Technical report, The SPIRIT Consortium, March 2008. 27, 51

[SS04] O. Sinnen and L. Sousa. List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, January 2004. 62, 83, 91

[SS05] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, June 2005. 71

[SSRM94] C. Selvakumar and C. Siva Ram Murthy. Scheduling precedence constrained task graphs with non-negligible intertask communication onto multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):328–336, 1994. 62

[SSS06] O. Sinnen, L. Sousa, and F. E. Sandnes. Toward a realistic task scheduling model. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):263–275, 2006. 74

[Tex08] Texas Instruments. TMS320C6474 Multicore Digital Signal Processor. Technical report, Texas Instruments, October 2008. 49

[THW99] H. Topcuoglu, S. Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 3–14, 1999. 62

[THW02] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002. 62

[WG90] Min-You Wu and Daniel Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990. 58, 60

[WM89] Wing Shing Wong and Robert J. T. Morris. A new approach to choosing initial points in local search. *Inf. Process. Lett.*, 30(2):67–72, 1989. 61

[WSRM97] Lee Wang, Howard Jay Siegel, Vwani R. Roychowdhury, and Anthony A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *J. Parallel Distrib. Comput.*, 47(1):8–22, 1997. 61

[WYJ+04] A.S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 15(9):824–834, Sept. 2004. 61

[WYKH97] Sung-Ho Woo, Sung-Bong Yang, Shin-Dug Kim, and Tack-Don Han. Task scheduling in distributed computing systems with a genetic algorithm. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia '97*, pages 301–305, Apr-2 May 1997. 61

[Xil08] Xilinx. Xilinx dataflow tools. Technical report, ASTG Technical Memo, Programming Solutions Group, Xilinx, May 2008. 19

[YG93] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993. 56, 60

[YG94] Tao Yang and A. Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, Sept. 1994. 58

Index

Acc, 90
Actor, 17
Advanced architecture model, 48
AEST, 100
ALAP, 102
ALST, 100
ASIC, 12
ATLAS, 2

BDF, 33
BSA, 62
BSP, 10
BTDH, 61
BU, 62
Bus, 47

CAD, 8
CAL, 17
    CAL2C, 21
    CAL2HDL, 19
CCR, 89
CCS, 8
Clustering, 58
    Linear clustering, 58
Communication delay, 102
Communication node, 46
Communicator, 46
CP/MISF, 60
CPFD, 61
Critical child, 101
CST, 24
Cut-through, 64

DAG, 26, 34
DCP, 58
DDF, 34
DFG, 6
DFS, 35
Distributed memory architecture, 44
DLS, 60, 62
DRT, 68
DSC, 58
DSH, 61
DSP, 1

EDK, 8
ELF, 9
Embedded system, 1
ETF, 60
EZ, 58

FAST, 61
FDDG, 12
FIFO, 11, 47
Flynn's taxonomy
  MIMD, 43
  MISD, 43
  SIMD, 43
  SISD, 43
FPGA, 1
FSL, 10, 11

GAUT, 14
GEF, 23
Graphiti, 23
GraphML, 26

Hardware/software co-design, 5
HDL, 5
HLF, 59
HLFET, 60

IDE, 9
IP, 13
IP Coprocessor, 46
IP-XACT, 51
ISE, 8

KPN, 31

List scheduling, 59
  Dynamic list scheduling, 59, 85
  Static list scheduling, 59, 82
LogP, 30
LST, 62

M4, 8
MCP, 60
MD, 58
Memory, 46
Message passing architecture, 44
MH, 62
MicroBlaze, 2
ML402, 20
ModelSim, 16
MPEG-4, 20
MPSoC, 2
multi-MicroBlaze, 8

Network, 17
Nios, 2
NoC, 12
Node duplication, 60
Node level
  Bottom level, 38
  Computation bottom level, 38
  Computation top level, 38
  Input bottom level, 78
  Input top level, 78
  Input/output bottom level, 80
  Input/output top level, 80
  Output bottom level, 79
  Output top level, 79
  Top level, 38
NP-hard, 12, 58

OpenDF, 16

Parallel embedded system, 2
Path, 35
  CP, 37
  Path length, 36
POSIX, 10
PowerPC 405, 2
PRAM, 29
PREESM, 3, 27

Processor, 45

Quartus, 8

RAM, 9
RAMP, 2
Rapid prototyping, 5
    AAA, 6
RISC, 1
Route, 64
Route step, 63
RTL, 14
RTOS, 10
RVC, 19

SAM, 9
SDF, 25, 32
SDF4J, 26
Shared memory architecture, 43
SocLib, 2
Store-and-forward, 64
SynDEx, 6
SynDEx-Ic, 12
SystemC, 21

Task scheduling, 56
TCP/IP, 10, 11
Time complexity, 95, 114
Topological Order, 35
Topology graph, 71

VHDL, 12
Visual Studio, 8

Xilkernel, 10
XML, 23
XSLT, 24

RESUME

L'architecture des ordinateurs est maintenant dans l'ère des multiprocesseurs permettant le calcul en parallèle. Les systèmes embarqués les plus récents s'appuient sur plusieurs processeurs DSP (Digital Signal Processor) ou MPSoC (Multiprocessor System-on-Chip). Corrélativement, les algorithmes des applications de traitement du signal et de l'image deviennent de plus en plus sophistiqués. La mise en œuvre de telles applications sur un système embarqué devient complexe. Aussi, les approches de prototypage rapide et de co-conception matérielle/logicielle sont souvent utilisées pour faciliter ce travail.

Le problème de l'ordonnancement des tâches, étape importante du prototypage rapide, est discuté et traité dans cette thèse. Nous cherchons des modèles d'ordonnancement des tâches en considérant précisément les communications entre les tâches. Nous modélisons ainsi l'algorithme de l'application comme un graphe acyclique orienté (Directed Acyclic Graph ou DAG), et nous proposons un modèle avancé décrivant de façon appropriée l'architecture du système embarqué parallèle. Après la formalisation du problème de l'ordonnancement des tâches avec ce modèle d'architecture, nous présentons plusieurs heuristiques d'ordonnancement basées sur la méthode de la liste (list scheduling) pour améliorer les performances de l'ordonnancement. Nos résultats expérimentaux attestent d'une accélération de l'application dans un contexte de moyenne ou de forte communication. Comme le poids des communications va en croissant dans les applications les plus récentes, que ce soient en communication numérique ou en compression vidéo, nos méthodes s'avèrent efficaces dans la mise en œuvre de ces applications sur systèmes embarqués parallèles. Nos méthodes d'ordonnancement sont intégrées dans PREESM, environnement de prototypage rapide basé sur Eclipse en "open source".

Mots-clefs: Prototypage rapide, système embarqué parallèle, ordonnancement des tâches

ABSTRACT

Computer architectures have come into an era of multiprocessor for parallel computing. Modern embedded systems also tend to consist of multiple processors like multicore DSP (Digital Signal Processor) or MPSoC (Multiprocessor System-on-Chip). Meanwhile, algorithms of signal and image processing applications become more and more complicated. Implementing such applications on a parallel embedded system with multiple heterogeneous components is not straightforward. Rapid prototyping and hardware/software co-design are usually used to facilitate this work.

The task scheduling problem is discussed in this thesis as an important step of rapid prototyping for developing parallel embedded systems. We aim at task scheduling models by accurately considering communications between computations. The algorithm of an application is modeled as a Directed Acyclic Graph (DAG) for task scheduling, and we propose an advanced architecture model to appropriately describe a parallel embedded system. After formalizing the task scheduling problem with the advanced architecture model, we also propose list scheduling heuristics with advanced techniques to improve the scheduling performance. Experimental results show that our methods usually accelerate an application in the case of medium or high communication. Since the communication cost is increasing in modern applications like digital communication and video compression, our advanced methods are suitable for efficiently implementing these applications on parallel embedded systems. Our methods are integrated in PREESM that is an Eclipse-based open source rapid prototyping framework.

Keywords: Rapid prototyping, parallel embedded system, task scheduling