Lehrstuhl für Integrierte Systeme
Technische Universität München

Methodology for System Partitioning of Chip-Level Multiprocessor Systems

Winthir Anton Brunnbauer

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. Ulf Schlichtmann
Prüfer der Dissertation:

1. Univ.-Prof. Dr. sc.techn. (ETH) Andreas Herkersdorf
2. Hon.-Prof. Dr.-Ing. Lajos Gazsi, Ruhr-Universität Bochum

Diese Dissertation wurde am 06.12.05 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 05.07.06 angenommen.

"He who can properly define and divide is to be considered a god."

Plato (427-347 BC)

Acknowledgements

The very beginnings of this work go back to the visions of Prof. Ingolf Ruge by bringing networking applications and marketing together. To be at the leading edge of technology, he arranged for the western outpost of the Lehrstuhl für Integrierte Schaltungen located at and funded by Infineon Technologies NA Corp., San Jose, CA, USA. I owe him special thanks for the opportunity to live and work in the Silicon Valley and be a part of the research community.

After my return from the USA, Prof. Ruge's successor, Prof. Andreas Herkersdorf, accommodated me at his institute and accepted the supervision of my PhD work. This knowledge and experience was a very valuable enrichment of this thesis. Thank you very much for fostering me!

For his offer to act as the second examiner, I am very grateful to Prof. Lajos Gazsi. His interest during the early stages of my PhD work inspired me that this topic is red hot.

I reminisce the "Munich Base" - the networking group of LIS with Thomas Wild, Nuria Pazos and Jürgen Foag - and my PhD buddy Gordon Cichon during my time at the LIS and Infineon. Unfortunately, he decided to change his plans and move back to Germany sooner than planned.

During and especially in the end of my PhD time, I appreciated the intense supervision of Thomas Wild. Thank you so much! Especially, I remember our daily field tests of video conferencing between two continents. Moreover, I owe special thanks to Verena Draga, Wolfgang Kohtz, Dr. Walter Stechele, and Doris Zeller for their support at LIS.

Many thanks to Charles Bry and Gunnar Hagen of Infineon Technologies NA Corp. for taking care of me through economy ups and downs in the Silicon Valley.

I have appreciated the endless discussions about all the world and his brother with Sven Haar, Roland Zukunft, Fabian Vogelbruch and Paul Zuber. I'm especially grateful to Fabian and Paul to be accompanied by them during the last steps of my PhD work.

A special thank you to my proof-readers Charles Bry, Anton Lindner, Hubert Mooshofer, Rainer Seidl, and Thomas Wild.

Many thanks to my parents, Otto and Margot Brunnbauer, for enabling me to pursue this doctorate. In particular, I am very much obliged to my wife Anja and Rainer Seidl for believing in me and motivating me during difficult times. This work is dedicated to them.

The financial support for this work by Infineon Technologies NA Corp. and Lehrstuhl für Integrierte Systeme is gratefully acknowledged.

Winthir Brunnbauer
München, December 2005

Abstract

The steady reduction of structure size in microelectronics allows the production of increasingly more complex and powerful integrated circuits. The increase of the productivity gap requires the development of new suitable design methodologies in order to increase the design productivity. By raising the entry level of automated design tools to higher abstraction levels, the usage of prefabricated modules permits a briefer and more efficient development of complex systems. Many design methodologies have been proposed which support only limited implementation possibilities based on the architect's experience. Hitherto, such kind of design tools can especially be found at research institutes.

The design of such integrated circuits, to find a suitable architecture meeting all specifications, is a challenging and complex task. Single chips comprising the same functionalities which have been implemented on boards are increasingly common, so-called systems on chip (SoC). The allocation of microprocessors and dedicated hardware on the SoC and the mapping of the different functionalities to hardware or software resources is a nontrivial task. In order to identify the best suitable implementation which meets the requirements, an extensive architecture exploration is necessary. Based on the ratio of software and hardware, the performance flexibility of SoC can be adjusted with respect to the application requirements.

This thesis addresses the partitioning of control dominated applications, in this case from the networking domain. Hitherto, only few algorithms are known which support this requirement during partitioning. Furthermore, the presented methodology partitions HW/SW systems with special focus on the internal communication of complex SoC architectures. This factor is increasingly recognized as a crucial factor for the efficiency of SoC. The necessity to use an efficient design of suitable communication architectures becomes important regarding potential system complexity.

A main contribution of this thesis is the extension of a constructive heuristic for the partitioning of functionalities with the consideration of events lying in the future. In particular, the consideration of inevitable imminent data transfers which may cause blockings of communication resources can result in improvements of the performance. The proposed variants of the improved partitioning methodology selects and assigns one task after another to suitable resources based on list scheduling. The application modeling carried out in this thesis utilizes annotated process graphs for the representation of functionalities and estimations of worst case execution times (WCET) for the resource library. With consideration of the current allocations of all resources, each task/resource combination which is ready to be partitioned is evaluated. The highest dynamic task priority decides about the selection of task and resource. A later change of the accomplished assignments cannot be performed. In this way, very short runtimes of the algorithms are possible even for large process graphs. However, these early assignments

may affect the overall performance unfavorably.

Partitioning algorithms usually handle each functionality of design models as self-contained which is individually assigned to a resource. Due to the granularity on the task graph, for instance, a memory access is represented as one functionality distributed in two nodes as design entity which is processed by one resource. However, partitioning algorithms may use different resources for these two nodes. In order to avoid such situations, a further contribution of this thesis is the enforcement of implementation constraints on multiple tasks. With the help of Common Implementation Nodes (CIN) grouped to clusters, various resources can be evaluated and one resource is assigned for all CINs of this cluster. This processing of design models permits a flexible consideration of design constraints.

On the basis of synthetic process graphs as well as a real-world example *DiffServ*, a networking application on a router linecard, the performances of the partitioning algorithms have been examined. Different scenarios of process graphs and target architectures allow statements about the behavior and applicability of the partitioning algorithms. The investigations of the variants of the constructive heuristic algorithms introduced in this thesis exhibit considerable improvements for wide ranges of parameters in relation to the conventional constructive algorithm. The provision for known inevitable events during task priority calculation allows a reduction in scheduling latency up to about 35%. However, the structure dependencies of the constructive algorithms on the process graphs and the content of the target architectures significantly influence the outcome. The improved algorithms cannot outperform the reference algorithm for all scenarios. To compensate for the structure dependencies within clusters, the usage of all CINs within a cluster for the priority computation instead of only the first CIN leads to better results in most cases.

The dependency of the constructive partitioning algorithms on the structure of the models becomes apparent with the evaluation of *DiffServ*. It has been shown that the performance of the introduced algorithms cannot be forecasted in advance. However, the short runtimes in the order of seconds allow the consecutive use of several variants of constructive algorithms to determine the best result. Thus, constructive partitioning heuristics can profit from the new approaches.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 System-Level Design

For the last years, embedded system design has been a very significant object of research. The development of electronic technology allows the integration of increasingly complex circuits forming complete systems on a common die. This has imposed the need for new techniques and tools to generate modern designs. In addition, the market pressure demands for shorter development times and cost effective solutions.

In [50] and [51], technology roadmaps for the design of integrated circuits (ICs) have foreseen systems including different types of processor cores and dedicated co-processors to fit on a single die, so-called systems on a chip (SoC). The assignment of software subprograms to multi-processor architectures is a nontrivial task. The design of distributed intelligent chip-level multiprocessor systems is a topic of intense research; for example [13], [21], [38], and [75].

In order to exploit these remarkable technological advances and be able to design such complex systems correctly, designers must be able to specify, refine, verify, and synthesize the components of an SoC correctly using design tools. Here, the gap between technology and tool maturity and acceptance is increasing instead of decreasing. Figure 1.1 shows this situation and the so called productivity gap: The number of available transistors grows faster than the ability to design them meaningfully. To counteract this issue, significant research activity is taking place in the area of embedded systems, especially in field of hardware/software co-design.

The rising number of available transistors increases the complexity of ICs. To cope with the number of design possibilities, a structured approach is required to produce the desired solutions. Design models help to reduce the complexity of the design problem to manageable structures by abstracting the representation of the components. Design models with different levels of abstraction are used during the design of ICs to refine abstract models toward the physical representation of the chip, [12]. [96] introduces a definition of

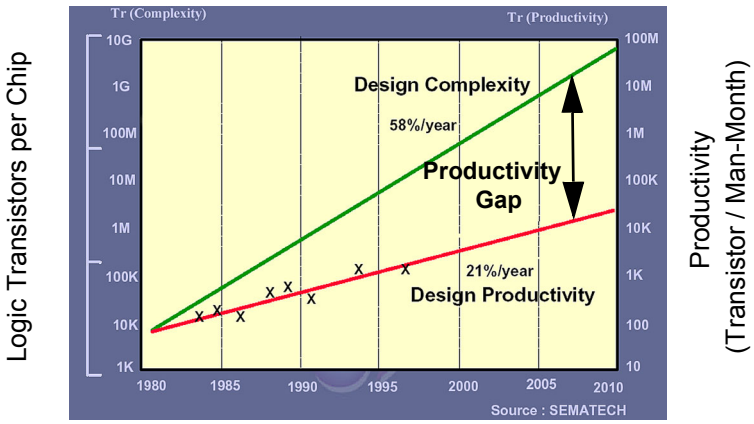


Figure 1.1: Design Complexity vs. Design Productivity by
the International Technology Roadmap of Semiconductors (ITRS)

a scale for taxonomy together with a classification of the different models used in IC design.

Design models represent different aspects of the circuits in the design flow of SoC. Some design models precisely specify the layout of the circuits and the arrangement of the transistors. Others estimate the power consumption which becomes increasingly important for SoC. Mobile applications are very dependent on the power consumption due to limited battery capacity. Even for server applications with permanent power supply, low power considerations are important to determine the heat dissipation of the devices.

Other design models separately represent the functionality and the implementation possibilities of the application in a very abstract form. Such design models may be used for the HW/SW codesign entry of SoC and consist of behavioral and structural objects. Behavioral objects comprise self-contained functionalities which may be derived from a verbal description or a C program. Structural objects are computation resources executing such behavioral objects.

The evaluation of design choices of ICs should move to the early phases of the design process to reduce the time needed for modeling and simulation. The abstraction pyramid according to [110] is illustrated in figure 1.2. The pyramid shows different levels of abstraction for the design models. At a given level of abstraction, different solutions may be explored and one solution is chosen. This selected solution is refined and acts as basis for potential solutions at a lower level of abstraction. Here, the refined design model allows different solutions to be explored, but also excludes the other solutions of the design space. To obtain the appropriate implementation of the SoC, the solution of the high level needs to be iteratively refined towards the lower levels by reducing the design space, [36]. The higher the levels of abstraction used for design entry, the more alternative realizations can be covered with the initial design model.

The cost of the model construction and evaluation is higher at the more detailed levels of abstraction. However, the opportunities to explore alternative realizations is

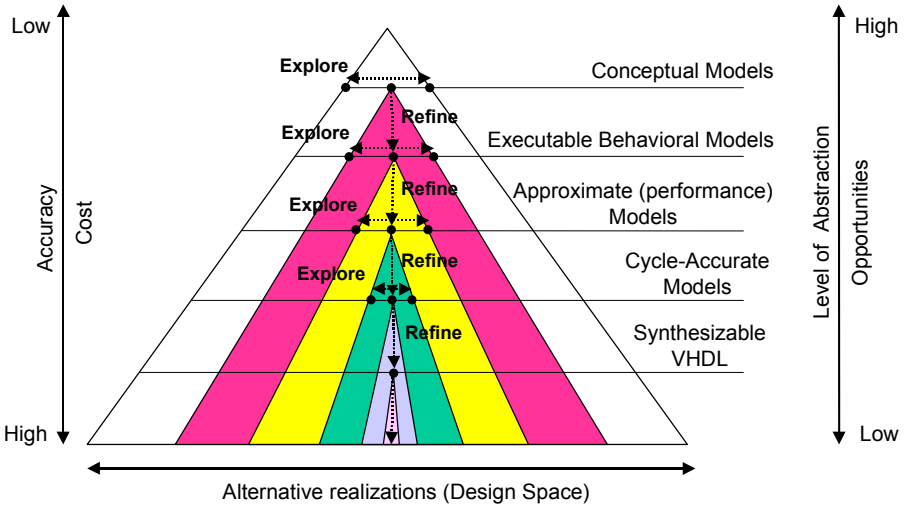


Figure 1.2: The Abstraction Pyramid According to [110]

significantly reduced at these levels. Hence, methodologies for modeling, simulation and design of embedded systems at higher level of abstraction are of special interest. The algorithms introduced in this thesis are located at the high levels of the abstraction pyramid to explore the potentials of the created architectures.

Levels of Abstraction

In the following, the levels of abstraction for the design of hardware (HW) components are introduced exemplarily. The overview of the levels of abstraction for HW design in table 1.1 and the description of each level is followed by the application of the design model to achieve productivity increase; [35], [37] and [89].

A description of embedded software (SW) development can be found in [81].

Levels	**Elements**	**Timing**	**Data**	**Language**
System	Blocks, Functions	No timing (WCET)	Abstract, Record	SystemC, C, Graph Representation
Architecture	Macros, Components	Cycles, Transactions	Abstract, Transactions	SystemC, HDL
Register/ RTL	ALU, MUX, Registers	Clock Cycles	Register values	HDL
Gate	Gates	Delay	Bits	HDL
Transistor	Transistors	Signal slopes	Signals	Spice, Layout

Table 1.1: Levels of Abstraction for HW Aspects and their Design Representations

- The transistor level describes models which consist of transistors which are the basic elements of integrated circuits. In the design models, these transistors are characterized by transistor equations and parameters. The layout of the transistors specifies the functionality of the IC. With circuit simulation programs, such as Spice, the circuit can be evaluated before it is manufactured.
- The gate level combines several transistors to handle logical functions, such as AND and OR bit operations, or more common, NAND and NOR bit operations. The models consists of netlists of gates. Gates are characterized by propagation delays, setup and hold times.
- The register level, also known as Register-Transfer Level (RTL), uses computational units containing logical functions, such as algorithmic and logic units (ALU), multiplier units (MUL), and registers storing values. In RTL models, a circuit can be described in hardware description languages (HDL), such as VHDL, [95], or Verilog, [47], as a set of registers and a set of transfer functions describing the flow of data between the registers.
- The focus of the architecture level is the allocation of components and their interaction for a given architecture. Transaction-level modeling (TLM) focuses on the communication as channels and transaction requests among the resources of the architecture. SystemC, [88], is designed to support TLM.
- The system level focuses on the selection of implementation including the choice of HW or SW for each behavioral object. The behavioral objects of the model are designed considering independent of the implementation. In this way, no implementation is preferred unintentionally. Graph based models suitably represent the application for such decisions. The properties of each resource are annotated to the model. System level design (SLD) tools support the designer in determining the most appropriate solution. Such SLD tools are mainly object of research at companies and universities.

Design models at system level are very suitable for the high level architecture exploration. The algorithms introduced in this thesis use system level design models to determine the appropriate implementation for each behavioral object.

Improvement of Productivity

The raise of abstraction levels over the last decades has enabled an improvement of productivity. Figure 1.3 gives an overview of the design entry levels of abstraction for starting designs and the automatization of lower levels over the last decades:

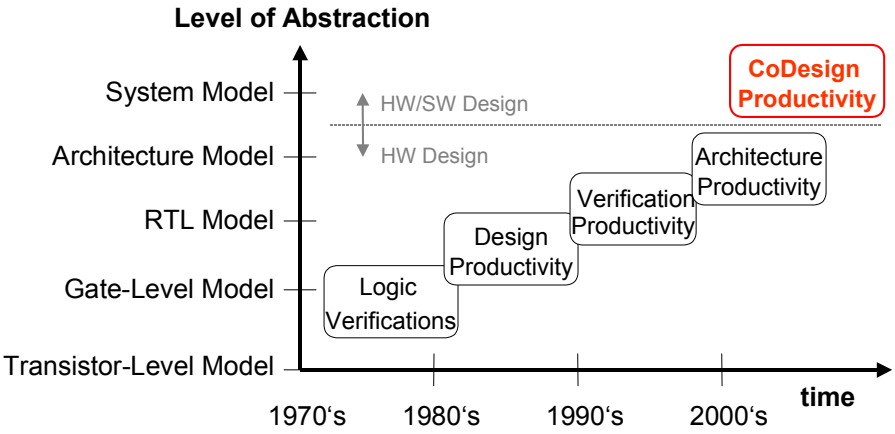


Figure 1.3: Abstraction Levels [4], [10]

The shift from the transistor level to the gate level has allowed logic verification. At the gate level, integrated circuits can be checked as to their correct logical behavior. In addition, the productivity of logic verifications is improved by focusing on the functionality instead of the physical design.

At RTL, data is processed as register values instead of single logical bit operations improving the design productivity. In HDL, the hierarchical structure of the design models thus facilitates the handling and processing of data. Design verification improves the overall yield of the IC design. Incorrect designs can be detected rather early. Therewith, the verification productivity is enhanced.

The architecture level allows the evaluation of the interaction among the architecture components. With the help of Transaction Level Modeling (TLM), the sequence and interdependency of functions and the corresponding resource can be analyzed. In this way, alternative system implementations can be validated before the SoC is completely designed, manufactured and tested. Problematic solutions can be identified and counteracted in an early stage of the design flow. The architecture productivity is improved by the usage of already designed blocks which can be reused for other designs.

At System Level, the behavioral objects are modeled independent of specific implementations. The selection of HW or SW should not be biased by specific modeling constraints. SLD tools map behavioral to structural objects and evaluate potential architectures. Supplementary changes of mapping decisions can be accomplished easily without costly modifications of the entire design model. The codesign productivity can be increased by evaluating more alternative realizations at high levels of abstraction than at lower levels. Hence, a high level of the abstraction pyramid should be chosen for the design entry.

However, each application domain has certain computation characteristics which need to be taken into account in the design model at system level. For instance, networking applications demand support of conditional branches to apply alternative networking protocols on data packets, whereas video applications require support of iterations and

loops to allow the execution of video processing algorithms. For different processing requirements, the appropriate models of computation, needs to be selected, [53].

Generic tools which can cover all application domains are difficult to extract. For example, [11] a commercial tool was withdrawn from the market, because it covered only specific application domains. The research activities in this area are located at research groups and universities.

In this thesis, networking is selected as the applications domain. The design of new devices such as network processors with multiple processor cores is challenging. SLD tools can support the designer in finding the suitable solution. This thesis focuses on specific SLD methodologies supporting the type of networking applications which are described next.

Application Domain Networking

The Internet Protocol (IP) has been established to communicate with computers around the world. This development highly influenced the way of information distribution and communication. In the first ten years, research was mainly pursued for the development of hardware equipment of the Internet backbone, the high-speed network spanned around the world. Figure 1.4 gives an impression of the network structure of the Internet. With increasing bandwidth available for the end-user, new end-user applications have shifted the focus towards the edge of the network. Rich multimedia, Voice-over-IP, and secure data transmission are a few applications which have more challenging performance and latency demands than previous applications.

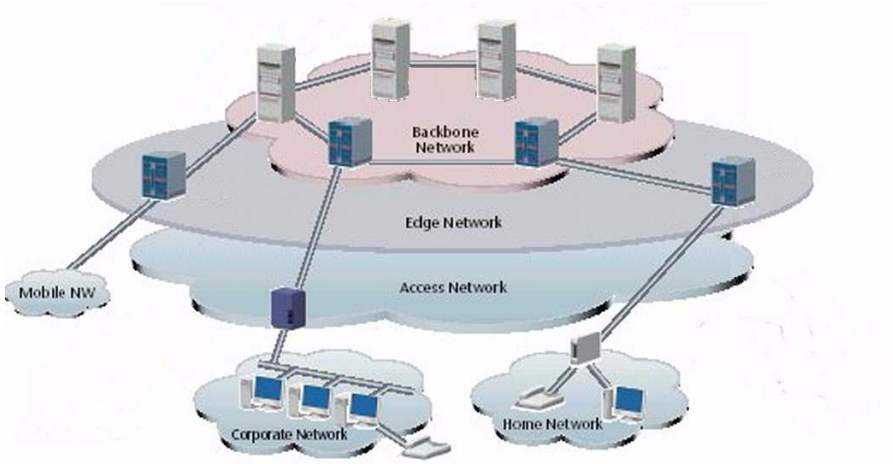


Figure 1.4: A Schematic Representation of the Internet

Network processors are mainly designed to be applied in the edge of the network and to flexibly support new applications. It is a device similar to a microprocessor, except that it has been optimized for use in applications involving network routing and packet

processing. The synthesis of the design and the evaluation of the internal processing of such network processors can be performed with the help of SLD tools.

Networking devices, such as Internet IP routers, comprise many functions which are triggered depending on the incoming data. The various types of utilized network protocols define the rules describing the transfer of data and communication between devices within a network. For the analysis of such applications, conditional branches are essential to cover the applied protocols in the design model. Networking as a control-dominated application necessitates the differentiation of different execution branches within the design models. Hence in this thesis, only SLD methodologies are considered which support conditional branches.

1.2 Design Flow

IC design is intended to result in a piece of working silicon. Starting a design with high level design models, a methodology is needed to transform them to lower levels of abstraction towards the physical representation of the circuit.

Teich introduced in [89] and [90] a synthesis-oriented model called double roof. One side of the roof represents those abstraction layers that are typically encountered in hardware design, whereas the other side represents those layers that are typical to software synthesis for embedded systems, see figure 1.5.

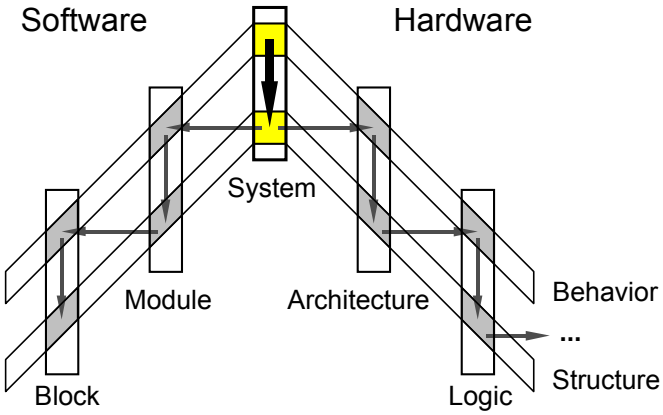


Figure 1.5: Double Roof Model of Teich

The upper roof represents the behavioral layer, whereas the lower roof stands for the structural layer. According to Teich, synthesis is the refinement of a behavioral specification into a structural specification at a certain abstraction level. The main synthesis tasks are independent from the level of abstraction and may be classified as

- allocation of resources,

- binding behavioral objects to allocated resources, and
- scheduling of behavioral objects on the resources they are bound to.

In each synthesis step, the allocation provides potential resources for the execution of behavioral objects or tasks. The binding process assigns tasks to resources. The scheduling determines the sequence of the tasks and occupancy of the resources. The resulting schedule finally indicates the performance characteristic of the considered architecture.

The successive refinement of the design models is performed towards the lower levels of abstraction to design complex HW/SW systems. Starting with a behavioral description of the system, system synthesis determines the complete system to be designed on the level of networks of communicating subsystems (e.g., processors, ASICs (Application Specific Integrated Circuit), dedicated hardware units, memories, buses, etc.), each realizing a part of the behavioral system specification (e.g., algorithms, tasks). The improved algorithms of this thesis can be located in figure 1.5 as the arrow transforming the behavioral system model to the structural system model.

This structural model of the system is refined and serves as the behavioral model of the HW architecture and the SW modules. Architecture models outline communicating functional blocks that implement coarse granular arithmetic and logical functions. Module models specify the interaction of function blocks that are mapped to a uni-processor or a multi-processor architecture in software.

Furthermore, the structure models of the architecture level and the module level are refined to serve as behavioral models for the logic level for HW and the block level for SW. Logic models contain netlists of logic gates and registers that implement Boolean functions and finite state machines. Block models typically include programs, functions, procedures as encountered in high-level languages that are refined to the instruction-level of the target-processor on which the code is to be executed.

HW/SW System Design Flow

As shown in Teich's Double Roof model, system synthesis determines the most appropriate implementation in HW or SW for each behavioral object. After these decisions are made, the refinements of the design are made at lower levels of abstraction. Figure 1.6 depicts a representation of the HW/SW system design flow.

The reference for all further efforts to design an IC is the specification. It is a clear, accurate, and detailed description of the objective and the technical requirements. Specifications usually contain among other things the description of the desired functionality, the required performance, and the information about maximum power and area. Based on this specification, a design model of the application at system level is created. The resource library contains information about the characteristics of all potential resources. Common characteristics of the used resources are performance, power consumption, area, etc..

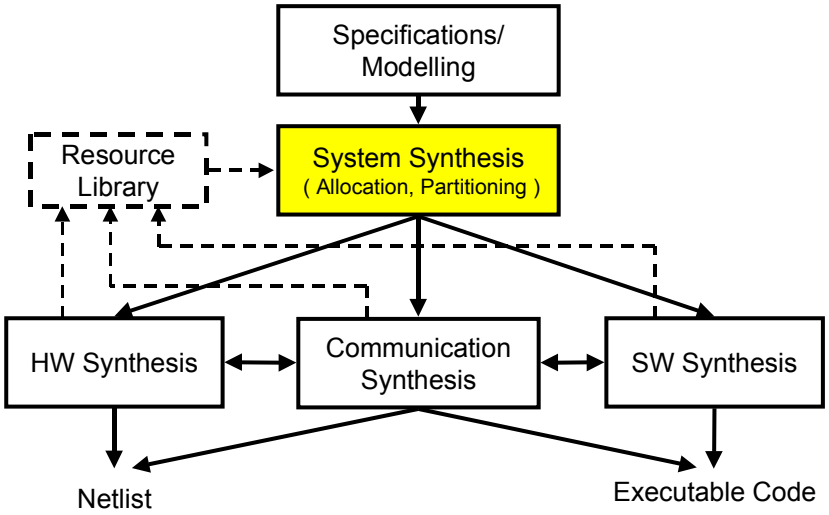


Figure 1.6: HW/SW System Design Flow, According to [3]

A paradigm shift of the SoC takes place regarding the scope of the specification. The specifications of integrated circuits have been contained single components with test rules which are produced and verified by system vendors. Nowadays, the specification increasingly defines the entire SoC and the responsibility for correctness is moved to the chip manufacturer. Hence, tools to support the revision of concept and specification are required.

System synthesis tools utilize design models and resource libraries and explore possible solutions by using allocation and partitioning. During allocation, a selection of resources is taken from the resource library. This allocation of resources is representing an architecture and called target architecture. Applied to partitioning algorithms, the target architecture comprises resources which may be used in the further design steps. The actual usage of the resources is determined during partitioning. The used resources of the target architecture form the SoC architecture. If the SoC architecture meets the requirements at this stage of the design, it serves as the basis for the hardware synthesis, the software synthesis, and the communication synthesis. Otherwise, the target architecture needs to be adjusted and partitioned again to explore the next potential architecture.

Communication is very important for the result of system synthesis. If communication is neglected during system level analysis, this disregard can result in unconsidered bottlenecks and a degradation of the performance. At system level, separating communication from computation is essential to cope with system design complexity, [59]. Design models need to be constructed without a specific implementation in mind in order not to restrict potential design opportunities at an early stage. However, communication is often linked with the behavior of the components so that it is very difficult to separate communication from computation. The occurrence of communication significantly interdepends with the system synthesis decisions. Considering communication to improve partitioning algorithms is the objective of this thesis.

In the following, the further synthesis steps are briefly introduced.

For HW synthesis, HDLs are used to describe the behavioral models. The used code may be written at a higher level than RTL level allowing to hide much of the complexity from the designer. The code can be synthesized into a low-level description for layout and analysis. The result is a netlist which is a version of an electronic circuit consisting of all of the circuit elements and their interconnections. This netlist is provided to the well established design flow, [86]. Commercial tool suites supporting HW synthesis can be found at [9] and [87].

The software synthesis is the process of taking a high-level description of functionalities in form of a software program and turning it into a lower-level description. The compiled program can be executed on the microprocessors of the target architecture. Examples for embedded software design tools are [80] and [103].

Communication synthesis defines the interfacing of hardware and software components. The aim of communication synthesis is to maximize efficiency of the communication resources and to minimize the communications overhead delays while respecting specification constraints. Gogniat et al., [41], propose an extended communication synthesis method. In [68] through [71], Lahiri et al. present a methodology and efficient algorithms for the design of high-performance system-on-chip communication architectures.

The results and experiences of the hardware, software and communication synthesis are fed back to the resource library. Thus, resource models can be described as precisely as possible.

System Synthesis

In [36], Gajski defines system synthesis as the process of converting the functional description of a black box into a structure of the black box that consists of connected components from a given library. During this process, several different design decisions have to be made. Such design decisions are the selection of components or the scheduling of computations and communications. The design process consists of design decisions causing model transformations to a structural model.

The evaluation of architectural decisions is depicted in figure 1.7 showing an architecture exploration loop during system synthesis. It consists of allocating a target architecture, partitioning the design model, and the evaluation whether the results are in compliance with the specification. The partitioning algorithm determines the binding of the behavioral objects to resources of the target architecture. In addition, a scheduling algorithm determines the sequence and the occupancy of each resource.

The architecture exploration loop varies the target architecture and the available resources in the case the requirements of the specification are not met. The evaluation of the results controls the architecture exploration loop and decides what changes in the target architecture and the algorithm constraints need to be performed.

In [35], Gajski et al. describe system partitioning as the procedure of implementing system functionalities on system components. The functionality of a system is

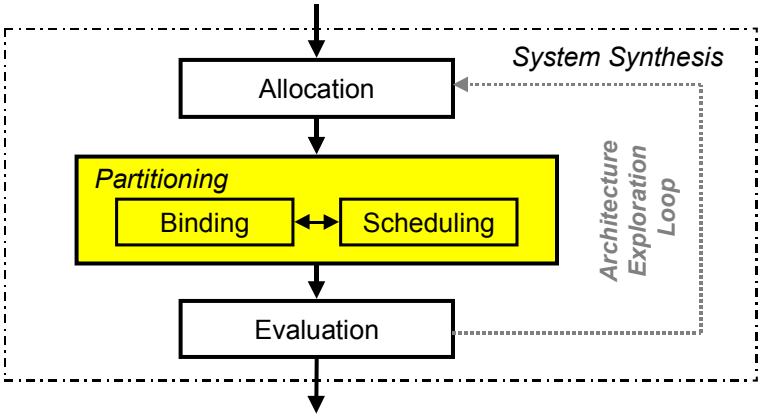


Figure 1.7: System Synthesis and Architecture Exploration Loop

implemented with a set of interconnected system components. In order to obtain such an implementation, the system designer must solve two issues: selecting a set of system components (allocation), and distributing the system's functionality among these components (partition). The allocation and partition must be chosen such that they will lead to an implementation that satisfies a set of design constraints, for instance pertaining to financial costs, performance, size, and power consumption given by the specification. Partitioning is a central system design task for SoC and the principle of partitioning will be exhibited in the following.

Gajski et al., [35], phrases the partitioning problem as:

> Given a set of objects $O = \{o_1, o_2, \ldots o_n\}$, determine a partition $P = \{p_1, p_2, \ldots p_m\}$ by assigning the objects to partitions $o_k \rightarrow p_h$ with $1 \leq k \leq n$ and $1 \leq h \leq m$ such that $p_1 \cup p_2 \cup \ldots p_m = O$, $p_i \cap p_j = \varnothing$ for all $i, j, i \neq j$, and the cost determined by an objective function *Objfct(P)* is minimal.

The objective function is an expression combining multiple metric values into a single value that defines the quality of a partition with a return value called cost. Since many metrics may be of varying importance, a weighted sum objective function can be used, e.g.:

$$Objfct(P) = k_1 \cdot area + k_2 \cdot latency + k_3 \cdot power = Cost \qquad (1.1)$$

By adjusting the coefficients k_1, k_2 and k_3 in eq. 1.1, the diverse results of the partitioning algorithm can compose a pareto curve, [38]. Solutions on a pareto curve can only be improved in one objective by degrading at least one of the other objectives. In this way, solutions for different objectives, such as high-performance or low power, can be found. The algorithms introduced in this thesis focus on performance as criteria for the object function. Other metrics can be considered in the architecture exploration.

Figure 1.8 shows an example of the interaction of allocation and partitioning with binding and scheduling. The design model of the application may be given as task graph. The introduction and description of task graphs are given in chapter 3. The target architecture is allocated and composed of resources of the resource library. During binding, behavioral objects of the design model are bound to resources of the target architecture. The scheduling process determines the sequence of the behavioral objects and the occupancy of the resources. The results of the scheduling may act as objective function for the binding selection.

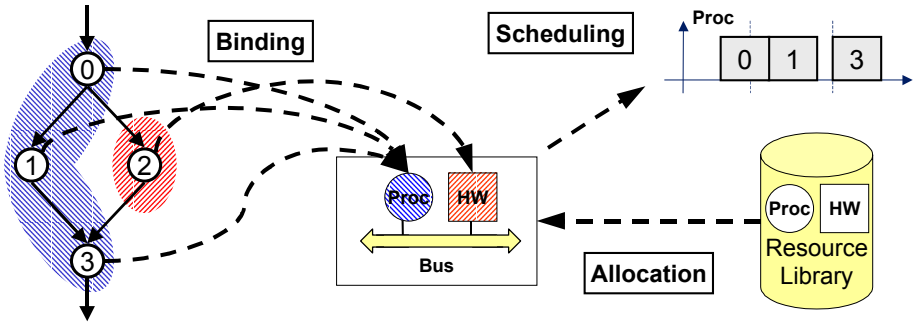


Figure 1.8: Combination of Allocation, Binding and Scheduling

The determination of the appropriate solution during system synthesis is a non-trivial task for partitioning algorithms. Concurrent execution of the behavioral objects and the number of resources significantly influence the number of possible solutions. The number of possibilities exceeds the mental capability of the designer to generate a solution meeting specific requirements simply by inspection.

The knapsack problem is an example which illustrates the scenario best. The term knapsack problem invokes the image of the backpacker who is constrained by a fixed-size knapsack. The knapsack may only be filled with the most useful items. Every item has a cost and value, so the most value for a given cost is sought.

The class of partitioning problems is known to be NP-complete; [14], [65]. All known algorithms for solving these problems have the characteristic that as the problem size increases, the number of steps necessary to solve the problem increases exponentially. These problems can be solved in polynominal time on a nondeterministic machine, but for which no deterministic polynomial time algorithm is known. To solve this problem, exact solutions can be found using integer linear programming. When exact solutions prove too costly to compute, heuristics can be used. Often the heuristic solution is close to optimal and meets the requirement of the specification.

1.3 Scope and Objective

The objective of this thesis is the determination of a suitable high-performance partitioning algorithm. Heuristic partitioning algorithms are very powerful at high levels of abstraction and therefore considered in this thesis. Such types of algorithms can be applied in the architecture exploration loop within the system synthesis design flow. The selected algorithm focuses on performance as criterion for the binding decision. Networking as application domain requires the partitioning algorithms to support conditional branches in the design model.

The consideration of more information during the selection of the appropriate binding and scheduling decision of the behavioral object can lead to an improved performance of the partitioning heuristics. The overall performance can be improved by taking into account future events caused by communication and the execution of behavioral objects. In this thesis, communication is used intensively during partitioning to avoid bottlenecks and the degradation of the performance.

Furthermore, the combined treatment of behavioral objects with similar characteristics allows the usage of different levels of detail in the design model. Different implementation realizations can be compared simultaneously. In addition, different design objectives of the architecture can be used within one single design model. Different design objectives can place special emphasis on the intended processing represented by the design model, such as parallel processing or the acceleration of certain functionalities. This combined treatment can reduce the modeling effort.

Approach

The selection of a suitable high-performance partitioning algorithm is the first step in this thesis. Since exact solutions are too costly to compute, heuristics are used in this thesis.

[101] shows that iterative heuristics have significantly longer runtimes than constructive heuristics. Supporting conditional branches for networking applications, the constructive heuristic of Xie et al., [107], is selected as basis partitioning algorithm in this thesis. On the basis for this task graph based algorithm of Xie et al., the Reference Constructive Algorithm (*ReCA*) is derived with modifications in the scheduling algorithm and the handling of conditional branches to allow the utilization of synthetic task graphs for performance evaluation of potential networking applications. To allow the comparison of the results, an ample number of different task graphs are necessary which represent the same characteristics. The task graph generating tool "TGFF: Task Graph for Free," [24], produces synthetic task graphs with similar characteristics regarding the structure of the task graph and the contents of the resource library.

Besides the short runtimes of constructive heuristics, [101] also shows that the results of constructive heuristics are worse than iterative heuristics. Communication is an important element of information during partitioning to improve the performance of partitioning algorithms. Hence, *ReCA* is improved by utilizing intensively events of communication and succeeding behavioral objects lying in the future. The improved partitioning algorithm is

called Enhanced Constructive Algorithm (*ECA*). Three variants of *ECA* with different considerations of future events are introduced. These various variants of *ECA* are compared to *ReCA* with the help of TGFF.

To facilitate the generation of tasks graphs for real-world applications, the clustering of Common Implementations Nodes (CIN) are introduced. These aggregated nodes within the task graph represent a self-contained functionality which can be executed exclusively by a specific set of resources of the target architecture. In this way, different implementation possibilities can be compared in one single design model. For the evaluation of this feature, composed task graphs are utilized to generate such clusters of CINs. Different task graphs generated by TGFF are put together to simulate structures of the task graph used in real-world applications.

Finally, a design model of a real-world networking application is used to evaluate the performance capabilities of *ECA*. This design model is applied to several variants of *ECA*.

1.4 Outline

Chapter 2 gives an overview of different partitioning algorithms. One of the presented algorithms is chosen to be the basis of the further work. In chapter 3, the Reference Constructive Algorithm (*ReCA*) is derived from the selected algorithm of chapter 2. The performance of *ReCA* is evaluated with the help of synthetic task graphs. Chapter 4 introduces the Enhanced Constructive Algorithm (*ECA*), the improvement of *ReCA*. The consideration of future events and the clustering of CINs are introduced. Chapter 5 applies *ECA* on a design model of a real-world networking application to show the effectiveness of the algorithm improvements. Chapter 6 provides conclusions and an outlook on future work.

Chapter 2

Related Work

This chapter introduces partitioning methodologies suitable for SoC design applied for system synthesis. Partitioning determines the implementation for all behavioral objects of a design model. The partitioning algorithms can be categorized in exact methodologies and heuristics. Exact methodologies can determine optimal solutions. However, these algorithms are very sensitive to the problem size. Representatives are enumeration and integer linear programming (ILP). With an increasing problem size, heuristics allow a faster runtime of the algorithm than exact methodologies by approximating the optimal solution. The heuristic methods differentiate between iterative and constructive algorithms. The iterative algorithms search the solution space, whereas constructive algorithm construct a potential solutions in a single step. Figure 2.1 depicts an overview of common partitioning algorithms..

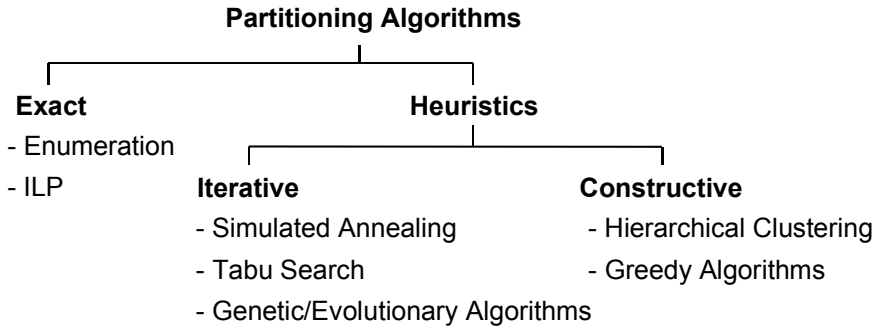


Figure 2.1: Types of Partitioning Algorithms

In this thesis, the considered application domain networking requires special provisions for the handling of conditional branches. This requirement is decisive for the selection of the selected partitioning algorithm.

2.1 Exact Methodologies

These methods include enumerative search techniques and approaches based on integer linear programming. Some examples should give a better understanding of the mode of operation of the algorithms.

2.1.1 Enumeration

A system is specified by a set of time-critical behavioral objects and constraints associated with each behavioral objects, such as timing. The assignment of the behavioral objects to resources of the target architecture decides about whether all given constraints are met.

The enumeration problem solving process starts with a feasible solution. Then, it sorts *Taken* and *Non-Taken* items in Non-decreasing weight order. The next step recursively examines all solutions by dropping and adding items in non-decreasing order. The enumeration algorithm can be illustrated by the subset sum problem, [52]:

- Minimize $\sum_{i=1}^{n} w_i x_i$ and subject to $\sum_{i=1}^{n} w_i x_i \leq c$ with $x_i \in \{0, 1\}, i = 1, \ldots n$, and w_i as the weight of the selected product.
- "Tolerance" level: If a feasible solution can be found that satisfies $\sum_{i=1}^{n''} w_i x_i < c + \varepsilon$, this solution is used and the exploration is terminated.

Applying the enumeration problem to the partitioning problem, x_i would be comparable to the usage of a specific binding with w_i as the corresponding resource characterization. c would be the timing constraint of the entire design. The following example should show the procedure of finding a solution. Out of a list of binding items with a specific characterization (weight), shown in table 2.1, a set of binding items should compose a minimum total weight of $c = 50$ and with $\varepsilon = 3$ a maximum total weight of 53:

Binding Item i	Weight w_i	Step 1	Step 2	Step 3	Step 4a	Step 4b	Step 5
1	25	x	x	x			
2	20					x	x
3	18	x	x		x	x	x
4	15		x	x	x	x	

Table 2.1: Example for Enumeration

Binding Item i	Weight w_i	Step 1	Step 2	Step 3	Step 4a	Step 4b	Step 5
5	14	x		x	x	x	x
Total	92	57	58	54	47	67	52

Table 2.1: Example for Enumeration

Start with binding item 1, 3 & 5 as taken (marks with 'x' in table 2.1), and 2 & 4 as not taken:

Step 1.	Set with item 1, 3 & 5	($\Sigma = 57$): Solution is bounded (> 53)!
Step 2.	Drop item 5, add item 4	($\Sigma = 58$): Solution is bounded!
Step 3.	Drop item 3, add item 4	($\Sigma = 54$): Solution is bounded!
Step 4.	Drop item 1, add item 4	($\Sigma = 47$): Infeasible!
	Additionally add item 2	($\Sigma = 67$): Solution is bounded!
Step 5.	Drop item 1, add item 2	($\Sigma = 52$): Solution found!
		Exploration terminated!

By trying out all possibilities, a solution may be found satisfying all design constraints. A more relaxed tolerance level may allow to find a solution sooner than using a strict tolerance level. However, an increasing number of binding items can cause the enumeration problem to be impractical to solve.

An application of enumeration for hardware/software partitioning is the methodology of D'Ambrosio et al., [20]. They use a tool based on enumeration to find implementations with forms a pareto curve. This approach suffers from long run-times and should only be considered if the estimation of costs and performance can be proven to be highly accurate, and if the number of objects is rather small.

Li et al., [72], examine the problem of determining the bound on the running time of a given program on a given processor. An important aspect of this problem is determining the extreme case program paths. The explicit enumeration algorithm of program path reaches its performance limit rather quickly since the number of feasible program paths is typically exponential in the size of the program.

2.1.2 Integer Linear Programming (ILP)

Linear programming (LP) is a technique for finding the best of all possible solutions of a system of linear equalities and inequalities describing design constraints. The criterion for the best solution is the maximum or minimum value of a given linear function of bounded variables, called the objective function. Such an objective function can include implementation characteristics regarding, for instance, performance, area, or power. Integer Linear Programming (ILP) restricts its possible solutions to integers in order to reduce the runtime.

An Integer Linear Program is a problem that can be expressed as follows:

- Minimize the objective function $\boldsymbol{cx}$
- subject to the constraints defined by $A\,\boldsymbol{x} \;\leq\; \boldsymbol{b}$

with $x_i \in \aleph$. $\boldsymbol{x}$ is the vector of variables to be solved for, A is a matrix of known coefficients, and $\boldsymbol{c}$ and $\boldsymbol{b}$ are vectors of known coefficients.

Geometrically, see figure 2.2, the linear constraints define a convex polyhedron, which is called the feasible region. Since the objective function is also linear, all local optima are automatically global optima. The linear objective function also implies that an optimal solution can only occur at a boundary point of the feasible region.

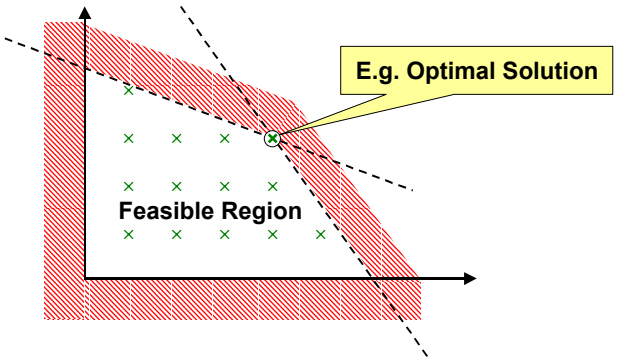


Figure 2.2: Feasible Region of ILP

There are two situations in which no optimal solution can be found. First, if the constraints contradict each other ($x \leq a \wedge x \geq a + 1$), then the feasible region is empty and there is no solution at all. In this case, the ILP is said to be infeasible. Alternatively, the polyhedron can be unbounded in the direction of the objective function ($x \geq a \wedge x \geq a + 2$), in which case there is no optimal solution since solutions with arbitrarily high values of the objective function can be constructed.

These constraints can model very general problems and can be solved as a sequence of linear programs. However, it can be extremely hard to solve. Most problems are solved using specialized procedures, such as branch and bound. Indeed, the most widely used method for solving integer programs is branch and bound. Subproblems are created by restricting the range of the integer variables. For binary variables, there are only two possible values: setting the variable to 0, or setting the variable to 1. More generally, a variable with lower bound l and upper bound u will be divided into two problems with ranges l to q and q+1 to u respectively. Lower bounds are provided by the linear-programming relaxation to the problem: keep the objective function and all constraints, but relax the integer restrictions to derive a linear program. If the optimal solution to a relaxed problem is (coincidentally) integer, it is an optimal solution to the subproblem, and the value can be used to terminate searches of subproblems whose lower bound is higher.

Theoretically, this can solve any ILP. Practically, it often takes lots of computational effort and time.

As an example for ILP during architecture exploration, Schwiegershausen et al., [83], presents a system level design methodology for the optimization of heterogeneous multiprocessor systems. It starts from specification of the application scheme, explores the design space based on a finite set of parameterized processor modules, and uses mixed integer linear programming as mathematical framework.

Arató et al., [1], use ILP to find a solution for a partitioning problem. The constraints are set to minimize the usage of hardware accelerators of a given target architecture while the overall performance is satisfied by using software and processors. The algorithm yields optimal solutions. However, the runtimes are very long even for small problems.

2.2 Heuristic Methodologies

Architecture exploration with the help of HW/SW partitioning is a very complex task. Exact methodologies can deliver optimal solutions only for small problem sizes in a practical time. For larger problems, heuristic methodologies offer the best trade-off, since the approximation of the solution towards the optimum can already satisfy the requirements without being optimal.

A heuristic is a technique designed to solve a problem that ignores whether the solution is provably correct, but which usually produces a good solution or even solves a simpler problem that contains or intersects with the solution of the more complex problem. Heuristics are intended to gain computational performance or conceptual simplicity potentially at the cost of accuracy or precision.

Partitioning heuristics can be categorized in two classes, iterative heuristic methodologies and constructive heuristic methodologies:

- Iterative approaches start from given partitions and then modify the partitions repeatedly with the intention of improving the partition result. The binding and scheduling are performed consecutively. Typical representatives of this methodology are simulated annealing, tabu search, and evolutionary approaches.
- Constructive approaches group the functions gradually into the available partition blocks until all functions are processed. A metric is used to evaluate the merits of assigning the functions into various blocks. Binding and scheduling are performed in a combined way. Representatives of this methodology are hierarchical clustering and greedy algorithms.

Iterative Heuristic Methodologies

Iterative approaches modify the bindings and then evaluate the changes of the results. For performance analysis, schedules are generated and form a criterion for further changes. This iteration is performed until either a desired constraint is met or the maximum number of iterations is exceeded. In the latter case, the outcome is the best solution out of all iterations.

2.2.1 Simulated Annealing (SA)

As its name implies, the simulated annealing (SA) exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system.

SA's major advantage over other methods is an ability to avoid becoming trapped at local minima. The algorithm employs a random search which not only accepts changes that decrease objective function, but also some changes that increase it. The latter are accepted with a probability

$$p = \mathrm{e}^{\left(-\frac{\delta f}{T}\right)} \tag{2.2}$$

where δf is the increase in f and T is a control parameter, which by analogy with the original application is known as the system "temperature" irrespective of the objective function involved.

Through equation (2.2), the annealing schedule determines the degree of uphill movement permitted during the search and is critical to the algorithm's performance. The principle underlying the choice of a suitable annealing schedule is easily stated - the initial temperature should be high enough to "melt" the system completely and should be reduced towards its "freezing point" as the search progresses.

Eles et al., [31], presents a heuristics for HW/SW partitioning of system level specifications based on simulated annealing. Results show that the performance and the runtimes of SA are outperformed by a tabu search based algorithm.

2.2.2 Tabu Search (TS)

The basic concept of tabu search (TS) as introduced by Glover, [39] and [40], is "a meta-heuristic superimposed on another heuristic. The overall approach is to avoid entrapment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited (hence ’tabu’)". TS proceeds according to the supposition that there is no point in accepting a new (poor) solution unless it is to avoid a path already investigated. This insures new regions of a problems solution space will be

investigated in with the goal of avoiding local minima and ultimately finding the desired solution.

A partitioning solution forms a local minimum in which always the same single change of task bindings is performed. To avoid using again the steps performed already, the method records recent moves in one or more tabu lists. The memory prevents the search from revisiting previously visited solutions and explore the unvisited areas of the solution space. In this way, oscillating behavior of the intermediate solutions can be prevented and other potential minima may be found.

In [62], Kwok et al. present a local search-based partitioning algorithm that attempts to design a scheduling algorithm of low complexity without sacrificing performance. The proposed algorithm is called Fast Assignment using Search Technique (FAST). The algorithm works by first generating an initial solution and then refining it using local neighborhood search.

A fast local search algorithm based on topological ordering is presented by Wu et al., [106]. This is a compaction algorithm that can reduce the schedule length produced by any DAG scheduling algorithm by determining the optimal search direction.

Wild, [101], introduces an iterative local search based method for mapping and scheduling of process graphs. For the local search, a new neighborhood definition is used which is based on the critical path of the graph. The corresponding extension leads to faster convergence and better results of the search.

2.2.3 Genetic/ Evolutionary Algorithms

Genetic or evolutionary algorithms incorporate the idea of a multi-point search strategy inspired from natural evolutionary processes. The motivation for this methodology is based on Charles Darwin's theory of the selection of better individuals, and Johann Mendel's theory of the generation of variants from selected individuals. Genetic or evolutionary algorithms are search algorithms based on the mechanics of natural selection and natural genetics. An iterative procedure maintains a population of structures that are candidate solutions to specific domain challenges, see figure 2.3.

During each generation the structures in the current population are rated for their effectiveness as solutions, and on the basis of these evaluations, a new population of candidate structures is formed using specific "genetic operators" such as mutation, inversion, and crossover to breed better models or solutions from an originally random starting population or sample, see figure 2.4. Mutation is a simple change in the structure, whereas inversion changes a segment of the structure. Crossover breaks and trades segments of different solutions with one another. The use of evolutionary techniques to diversify, combine and select options may improve performance, following the methods of natural selection by coding options as genes.

Evolutionary algorithms can provide a number of potential solutions to a given partitioning problem. The final choice is left to the user. Thus, in cases where the particular problem does not have one individual solution, for example a family of pareto-optimal

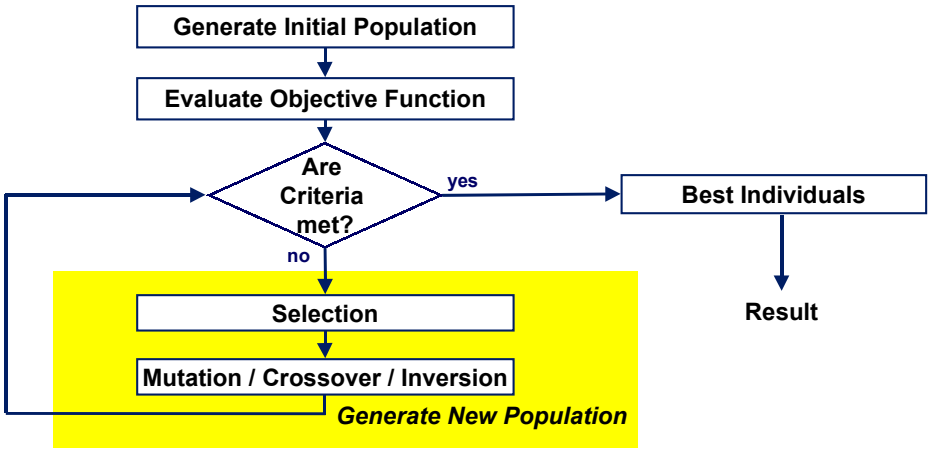


Figure 2.3: Evolutionary Algorithms

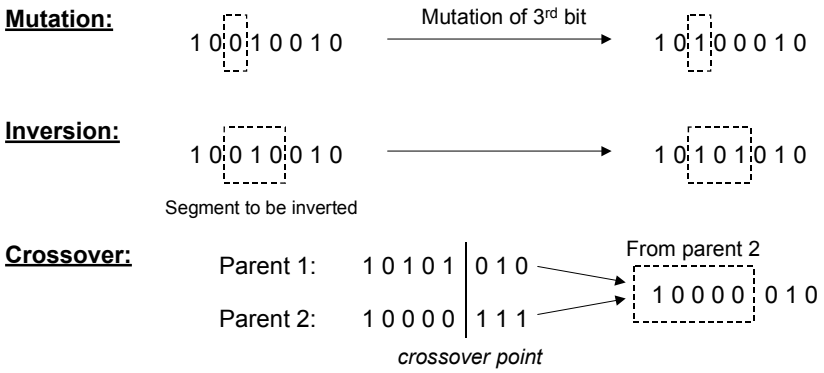


Figure 2.4: Genetic Operators on a Binary Coded Chromosome

solutions, as in the case of multi-objective optimization and scheduling problems, the evolutionary algorithm is useful for identifying these alternative solutions by exploring the design space.

Kwok et al., [63], propose an parallel genetic scheduling (PGS) algorithm. By encoding the scheduling list as a chromosome, the PGS algorithm can generate an optimal scheduling list which in turn leads to an optimal schedule. The genetic operators order crossover and mutation are used to improve the scheduling lists.

Blicke et al., [5], introduce an approach to system-level partitioning for data flow-dominant HW/SW systems based on an evolutionary algorithm. A graph-theoretic framework is used to describe algorithms, sets of architectures and user-defined binding constraints. The application of the design model is only shown for a small size of the specification graph. Larger graphs increase the runtime significantly.

Zitzler et al., [109], take [5], an evolutionary approach to multi-criteria optimization called the Strength Pareto EA (SPEA). Its reuse of non-dominated solutions during the execution can improve the performance of SPEA. Again, the graph size is rather small to gain results in a practical timeframe. Thiele et al. uses SPEA, [109], for analysis network processor applications in [15], [16], [17], [73], [91], and [92].

Dick et al. introduce different methodologies for different applications based on a genetic algorithm, [25], [26], [27], and [28]. Its multi-objective optimization generates pareto-optimal sets of architectures which trade off different cost. The scenarios introduced are limited in size of the task graphs to allow practicable runtimes for the analysis of the algorithm.

The iterative heuristics represent partitioning methodologies to find near-optimal solutions in a practical timeframe. However, the problem sizes are quite small with dozens of tasks to be partitioned. Constructive heuristics can generated solutions in a much quicker way and allow the processing of larger task graphs.

Constructive Heuristic Methodologies

Constructive partitioning algorithms start grouping the functions gradually into the partition blocks and result in a binding and scheduling solution. A metric is necessary to evaluate the merits of assigning the functions into various partition blocks. Constructive algorithms are able to deliver solutions in a quick manner compared to iterative algorithms. However, the immediate selection of tasks for the partition blocks cannot be rescheduled later on. Such selections may result in unfavorable choices for the overall design and may cause impairments in performance.

Hierarchical clustering and greedy algorithms represent two methodologies of constructive heuristics shown in the following.

2.2.4 Hierarchical Clustering

Hierarchical clustering (Johnson [55], Lagnese et al. [67], and McFarland et al. [74]) is a constructive method for partitioning which constructs initial partitions for the use of an iterative algorithm later on. The initial partitions are bound to the resources of the target architecture. The method groups pairs of behavioral objects based on a closeness metric between the objects. After each grouping step, the closeness metrics are updated. These partitioning tasks are repeated until a termination condition is met. For instance, if all objects are bound, or the closeness metrics of all objects drop below a given bound. The algorithm is fully characterized after defining the following issues:

- The closeness function that provides the proximity values.
- The cut level in the cluster tree that is built upon the closeness values.

Gajski et al., [35], describes an hierarchical clustering algorithm. The core of the algorithm compares the closeness metric of all nodes and merges the one with large values until all nodes are processed. Figure 2.5 shows a simple example with four nodes interconnected by edges whose closeness values between two nodes are annotated. The algorithm looks for the largest closeness value and merge the correspondent nodes.

Node 1 and 2 are subsumed to a cluster and the edges are adjusted accordingly. To approximate the closeness values between the new cluster and node 3 and 4, the average of the previous closeness values is applied. In graph I., the closeness value between node 1 and 3 is 25, and node 2 and 3 is 15. The adjusted value between node 3 and the new cluster is (25 + 15) / 2 = 20 in graph II. Out of the resulting graph, the largest closeness values is selected and the merging continues until only one cluster exists. The result (figure 2.5, IV.) is a cluster tree which maintains a history of the order in which the objects are merged to generate a variety of possible partitions.

A cutline, here in figure 2.5 (V.) arbitrarily drawn, can determine a partition of node 1 and 2 in one cluster and node 3 and node 4 each in another. The line can be drawn at any level of the cluster tree and also in a different angle. Each cluster may be bound to a resource of the target architecture. Numerous possible partitions can be generated by such lines, where each partition consists of groups whose objects are close to each another.

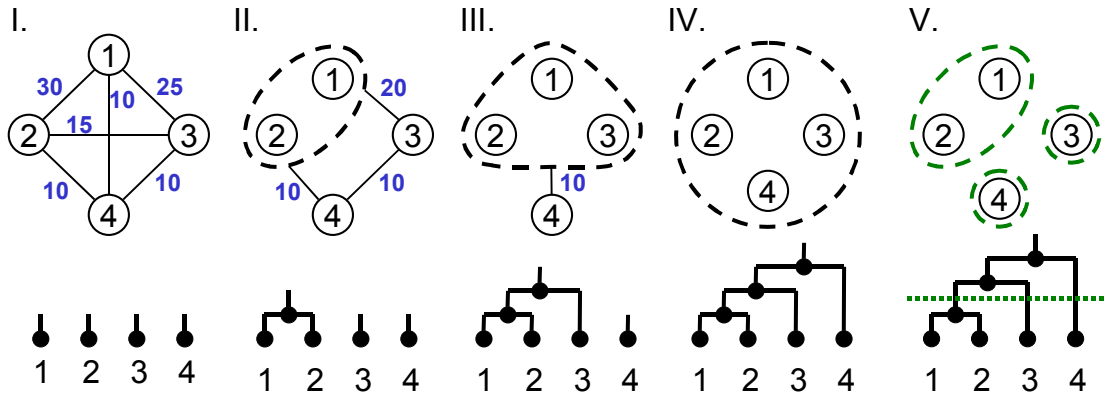


Figure 2.5: Hierarchical Clustering

2.2.5 Greedy Algorithms

Greedy algorithms always take the best immediate or local solution while finding an answer. Greedy algorithms determine the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for instances of other problems. If there is no greedy algorithm that always identifies the optimal solution for a problem, many possible solutions need to be searched to find the optimum. Greedy algorithms are usually quicker in processing, since they don't consider all details of possible alternatives.

Greedy algorithms are step-by-step recipes for solving problems. A greedy algorithm might also be called a "single-minded" algorithm or an algorithm that selects all of its favorites first. The idea behind a greedy algorithm is to perform a single procedure in the recipe over and over again until it can't be done any more and see what kind of results it will produce. It may not find the optimum solution, or, if it produces a solution, it may not be the very best one, but it is one way of approaching the problem and sometimes yields very good (or even the best possible) results in a short period of time.

Imagine a cashier who does not really consider all the possible ways in which to count out a given sum of money. Instead, the required amount is counted out beginning with the largest denomination and proceeding to the smallest denomination.

Suppose nine coins are given: five 1c (cent) coins, two 5c coins, and two 10c coins; $\{c_1, \ldots, c_9\} = \{1, 1, 1, 1, 1, 5, 5, 10, 10\}$. To count out 16 cents, we start with a 10c coin, then add a 5c coin followed by a 1c coin. This is a greedy strategy because once a coin has been counted out, it is never taken back. Furthermore, the solution obtained is the correct solution because it uses the fewest number of coins.

Unfortunately, greedy algorithm does not always produce the correct answer. Consider what happens if a 15c coin is introduced. Suppose a change of 20 cents is asked to count out from the following set of coins: $\{1, 1, 1, 1, 1, 10, 10, 15\}$. The greedy algorithm selects the 15c coin followed by five 1c ones - six coins in total. Of course, the correct solution requires only two coins provided the minimal number of coins is required. The solution found by the greedy strategy is a feasible solution, but it does not optimize the objective function.

The selection of the appropriate partitions and resources can be compared to the example above. Heuristic list scheduling algorithms perform such operations of greedy partitioning algorithms. They are widely used for their low-complexity and good performance. Usually, performance is taken as the optimization function for partitioning. Within architecture exploration, the main tasks of such constructive list scheduling algorithms are the following three items:

- The identification of the potential tasks to be processed next.
- The selection of the most appropriate task/resource binding.
- The proper scheduling of the task.

List Scheduling manages a list of task which need to be scheduled, both the sequence of consumed time slots in the schedule and the sequence of selecting tasks. This list is ordered by priorities for each task which are the metric for the selection decision. The higher the task priority, the earlier the task is to be scheduled. Figure 2.6 depicts the flow chart of a list scheduling algorithm.

After the global priorities of each task are determined, all tasks are identified which are ready to be scheduled next and taken into account for the selection of the next binding. Originally, list scheduling algorithms determine the priority of all nodes before the scheduling takes place. This selected sequence of the nodes is fixed and ordered by static

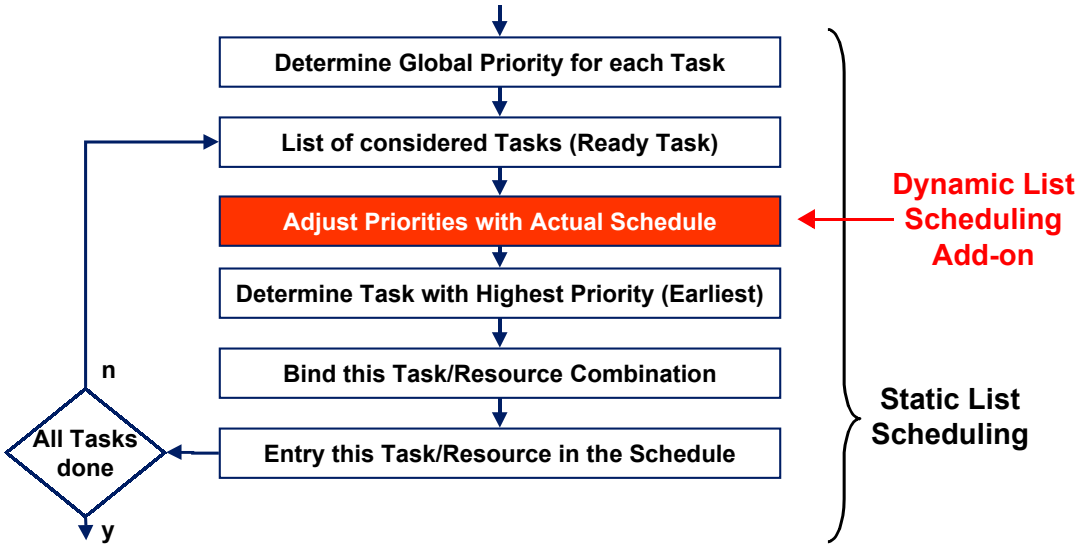


Figure 2.6: List Scheduling Algorithm

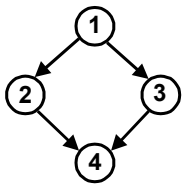
priorities. Even if an unfavorable situation arises during the scheduling, a change in priorities is not possible. The usage of the current schedule for priority updates during partitioning allows an improvement of the performance. The situation based selection allows the choice of the most favorable task. In this way, dynamic priority calculation can cope with congested resources, especially communication resources.

However, a wrong decision in the sequencing of the tasks here can impair the overall performance significantly. A task scheduled earlier with the highest priority at the time of selection may block valuable resources and extend the total latency. When the selection is done, the task is bound to the resource and no rescheduling is possible to fix issues later on in the scheduling. A change of the binding and scheduling sequence here would require to start the partitioning all over again.

With only a small set of tasks to be examined, fast analysis times can be achieved. However, the binding decisions have to be done with only partial knowledge of the total schedule. Therefore, the selection process of the task is very critical to the overall performance.

Figure 2.7 shows the priorities of the single tasks and the sequence of binding and scheduling (step A through D) with the help of a small example. The tasks 1 through 4 are to be partitioned with the data dependency depicted in the task graph. With all four task to be bound and scheduled, task 1 has the highest priority to be selected in step A. Task 2 is planned to be selected next. After the partitioning of task 1 in step B, the priorities are changed of the other tasks. Instead of task 2, task 3 is chosen next due to the consideration of the current schedule.

Such changes of the task priorities allow the response to unexpected congestions of resources. Especially, hardware accelerators are planned to be selected regardless of the communication resource. With heavy data traffic on the shared bus, large latencies can



Task Rank				
Task	Partitioning Step			
	A	B	C	D
1	1st	✓	✓	✓
2	2nd	2nd	1st	✓
3	3rd	1st	✓	✓
4	4th	3th	2nd	1st

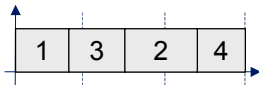


Figure 2.7: List Scheduling Example

occur which may impair the overall performance. In step C and D, task 2 and 4 are partitioned.

Dynamic-list scheduling algorithms, such as the Force Directed scheduling (Paulin et al. [77], Dutta et al. [30]), are able to utilize more information for the selection of the binding, e.g. for performance objectives, the schedule is updated after each binding.

In [58] and [57], Kalavade et al. introduce an algorithm with an adaptive objective mechanism based on a combination of global and local measures. Contradictory objectives, such as hardware area minimization and latency constraints, are taken into account by a global time-critical measure which selects an objective function in accordance with feasibility. Local characteristics of the nodes are considered by classifying nodes into local phase types.

Sih et al., [85], presents a compile-time scheduling heuristic called "Dynamic Level Scheduling", which accounts for inter-processor communication overhead when mapping communicating tasks onto heterogeneous processor architectures with limited or possibly irregular interconnection structures. This technique uses dynamically-changing priorities to match tasks with processors at each step, and schedules over both spatial and temporal dimensions to eliminate shared resource contention.

Xie et al., [107], extend this approach of Sih et al. and introduce a partitioning algorithm that efficiently handles conditional execution in multi-rate embedded system. They propose a mutual exclusion detection algorithm that helps the scheduling algorithm to exploit the resource sharing. Partitioning is simultaneously performed to take advantage of the resource sharing among those mutual exclusive tasks. This algorithm is used, inter alios, by Shin et al., [84], Wu et al., [105], and Vallerio et al., [94].

2.3 Features of Partitioning Algorithms

In this section, special aspects of partitioning algorithms are presented which increase the applicability and the performance of the algorithms.

2.3.1 Consideration of Communication

The consideration of communication is an very important factor of SoC design. Simultaneous accesses on shared media can block resource connections and can result in degradation of the performance. In particular, applications which are sensitive to latency increase require special consideration of communication. In the following, some examples of research work are presented which include communication during partitioning:

Kwok et al., [66] propose an algorithm that schedules the tasks as well as the communication traffic by treating both the processors and communication links as important resources. The algorithm adapts its tasks scheduling and mapping decisions according to the given network topology. Such as tasks, messages are also scheduled and mapped to suitable links during the minimization of the finish times of tasks.

In [46], Hu et al. present an algorithm which statically schedules both communication transactions and computation tasks onto heterogeneous Network-on-Chip (NoC) architectures. In NoC architectures, a chip consists of contiguous areas which are physically isolated from each other but have special mechanism for communication among each other. This work formulates the problem of concurrent communication and task scheduling for heterogeneous NoC architectures.

2.3.2 Support of Conditions

Control-dominated applications, for instance, of the networking domain, necessitate the differentiation of different execution branches. The support of conditional branches is essential to model and simulate such applications in a realistic way.

Consideration of Conditions by Explicit Set of Conditions

Eles et al., [29], [32], and [33], present a list-scheduling algorithm for task graphs with data and control dependencies. Its target is to derive a worst case delay by which the system completes execution, such that this delay is as small as possible. For each set of conditions, individual subgraphs and schedules are created which blank unused tasks. In this way, a customized schedule for each condition set can be derived. However, the effort to generate schedules for each condition set can be become quite large.

Consideration of Conditions by Mutual Exclusiveness

At high levels of granularity, the primary objective of the partitioning algorithm can be to derive an estimation of overall latency in worst conditions. Hence, the critical path of the workflow represents the most delayed execution. To consider all possible condition situations, a tedious analysis of possible execution scenarios may be performed. However, only the worst performing condition set will be considered.

To reduce analysis complexity, the selection of a conditional branch may be compared to a switch-case instruction. In this way, only a single branch is selected and active at a

given time. With only one single branch active, such conditional branches exclude each other mutually. Mutual exclusion allows the consideration of two or more independent tasks to be assigned to the same resource for the same timeframe. By considering all branches of one conditional fork simultaneously, the critical path can be identified.

Wakabayashi et al., [97], present a global list scheduling method based on condition vectors. The proposed method exploits a more "global parallelism" which parallelizes multiple nests of conditional branches and optimizes across the boundaries of basic blocks.

Juan et al., [56], propose an algorithm for comprehensive identification of mutually exclusive operators to improve the quality of the design independent of description styles.

Xie et. al, [107] proposes an algorithm for supporting conditional branches within CPGs, which detect mutual exclusion with the help of a branch labeling method. In this way, mutual excluded tasks can be identified easily. However, only structured branch models are supported.

Wild, [101], developed an approach which enables the treatment of graphs with control dependencies which are common for the processing of data communication protocols. A new procedure for the detection of the mutual exclusivity of graph nodes is defined. In the scheduling step, this information is used for the efficient usage of the resources. This extension is used for the partitioning algorithms in this work.

2.3.3 Look-Ahead

The constructive greedy algorithms shown in the previous sections perform partitioning in a very quick manner with the side effect of non-optimal results. A further design step of the dynamic list scheduling may be the consideration of future events.

Kwok et al., [61], propose a static scheduling algorithm for allocating task graphs to fully connected multiprocessors called the Dynamic Critical-Path (DCP). It determines the critical path of the task graph and selects the next node to be scheduled in a dynamic fashion. The schedule is rearranged on each processor dynamically in the sense that the positions of the nodes in the partial schedules are not fixed until all nodes have been considered.

Winckler, [102], proposes a dynamic decentralized look-ahead scheduling algorithm and a cooperation protocol. The information is utilized about the internal job structure concerning future service requirements and system state information for dynamically arranging schedules such that jobs can take advantage of inevitable waiting times of others.

Kalavade et al., [58] and [57], introduces a heuristic, called GCLP (Global Criticality, Local Preference). The heuristic reduces the greediness associated with traditional list-scheduling algorithms by formulating a global measure. The global measure also permits an adaptive selection of the optimization objective at each step of the algorithm.

2.3.4 Clustering

Clustering is employed in heuristics to reduce the complexity of partitioning. Functionalities are grouped to several clusters where each cluster is handled as one big behavioral object.

With the term flexibility, Taubelt et al., [45], introduce a design dimension of an embedded system that quantitatively characterizes its feasibility in implementing not only one, but possibly several alternative behaviors. A hierarchical graph model is introduced that allows to model flexibility and cost of a system formally with the help of clustering.

Cirou et al., [18], apply the clustering technique for heterogeneous systems and present the algorithm "triplet". They use an effective scheduling technique which consists in grouping tasks on virtual processors, called clusters, and then mapping clusters onto real processors.

Dave et al. present a hardware-software co-synthesis technique for real-time distributed embedded systems in [22] and [23]. The algorithm uses a new dynamic task clustering technique which takes the dynamic the critical path and the existence of multiple critical paths in the task graph into account during partitioning.

2.4 Summary, Comments and Conclusions

In this chapter, various partitioning algorithms are presented which may be applied in system synthesis. Exact and heuristic methodologies are introduced to select an algorithm for this thesis.

Exact methodologies, such as enumeration and integer linear programming (ILP), can determine optimal solutions. The solution approaches of exact methodologies are very time-sensitive to the problem size. Heuristics allow faster execution than exact methodologies by evaluating a part of the entire solution space. Hence, the results can only approximate the optimal solution. Iterative and constructive heuristics represent two partitioning algorithm types. Although iterative heuristics provide near optimal results, the execution of the iterative algorithms can take quite long compared to constructive algorithms. However, constructive algorithms may produce impaired results due to their task-by-task partitioning approach. A comparison of iterative and constructive heuristics is shown in chapter 3.

The fast runtimes of constructive heuristics allow the designers to obtain solutions immediately. To counteract the impaired performance of constructive heuristics, potential modifications of the task priority calculation are very promising to improve the results. In this thesis, a constructive heuristic is selected to serve as partitioning algorithm.

With the support of conditional branches, the methodology of Xie et al., [107], offers the most favorable base to use for further examinations. The methodology supports the abstraction level of system level design by having a task graph as functional representation

and annotated WCETs for architecture characterization. In addition, the constructive heuristics supports conditional branches necessary for networking applications, and utilizes heterogeneous resources for the target architecture. A detailed description of the algorithm is given in the next chapter.

Chapter 3

Reference Algorithm

In the previous chapter, a variety of potential partitioning methodologies are introduced. The constructive heuristics are considered in the following, since they provide a much faster execution compared to exact algorithms or iterative heuristics. With respect to the necessary support of conditional branches used in the modeling of networking applications, the partitioning algorithm of Xie et al., [107], is selected as the basis of the used partitioning algorithm in this thesis.

For performance evaluations, this algorithm is applied to synthetically generated design models in a sufficient quantity to allow statements about the performance. TGFF, a tool from Princeton University, [24], allows the creation of randomly generated and reproducible task graphs. Since the structure of such synthetic task graphs is random, a more improved handling of conditional branches is required than available in the algorithm of Xie et al. to apply TGFF. For this reason, a Reference Constructive Algorithm (*ReCA*) is introduced based on the algorithm of Xie et al. and the improved support of conditions of [101].

ReCA acts as a reference of the introduced improvements to *ReCA* shown later in this thesis. In the following, the way of modeling possible applications with the help of conditional task graphs (CPG) is presented. After the algorithm of Xie et al. is illustrated, the modification for *ReCA* are introduced. The partitioning algorithm is enhanced by more flexible target architecture, more flexible communication structure, and the optimization of the scheduling regarding performance. Subsequently, the performance of *ReCA* is compared against well known iterative heuristics using tabu search for binding changes and list scheduling for scheduling.

3.1 Modeling

In [35], Gajski et al. present some of the characteristics most commonly found in conceptual models used by designers. The main characteristics are concurrency, state transitions, hierarchy, programming constructs, behavioral completion, communication, synchronization, exception handling, non-determinism, and timing. It is noted that different classes of systems require specific subsets of these characteristics.

For embedded systems, the behaviors are defined by their interaction with their environment. This interaction can be performed by sequencing between a set of modes. Each mode may represent the current state or some computation. Such systems are constantly responding to external events and computing their outputs as a function of their inputs and their current state. Telecommunication systems are examples of embedded systems. For the modeling of such embedded systems, the following characteristics need to be considered:

- Behavioral Hierarchy: The process of decomposing a behavior into distinct sub-behaviors, which can be either sequential or concurrent, may be represented as either a set of procedures or a state machine.
- Non-Determinism: Within design models, more than one possibility for a particular transition or task can be available. Not to limit the system by a made choice, the designer can specify multiple options for the processing of transitions and tasks. For instance, functional selections in a program sequence, which will be chosen during run-time, describe a non-determinism which can be used to model conditional branches.
- Concurrency: The support of concurrent behaviors of the design model is important to allow parallel execution streams. Also, the concurrent behaviors need to cooperate with each other in order to achieve the functionality intended by the system as a whole.
- Synchronization of Execution Streams: At convergent branches, it needs to be ensured that all preceding tasks are completed before the succeeding task can be executed.
- Behavioral Completion: It is important to define states of behavioral completion for performance comparability of different solution approaches and to avoid infinite loops within design models.

Task graphs in design models are very suitable at representing the problems of system synthesis. The characteristics of task graphs meet the requirements for embedded systems mentioned above. Hence in the following, the design models are represented with the help of tasks graphs.

The structures of task graphs specify the data and sequence dependencies of the functionalities. The target architectures are characterized by WCETs (Worst Case

Execution Times) as annotations of this task graphs. The communication architectures of the design model are represented both in the task graphs by the graph vertices and in the target architectures by communication structures.

Representation of Functionality

The functionalities of the considered application can be composed in a conditional process graph (CPG), Eles et al., [29], [32], and [33]. A directed, acyclic and polar task graph represents the dependencies of the system functions. The behavioral objects are represented as graph nodes. To determine behavioral objects, the split within the whole functionality is made, where communication is either necessary or reasonable, or where the functionalities build conclusive units. The communication may be caused by either switching over to another computational resource, or loading necessary data from the memory. The communication or data transfers between the components of the target architecture can be modeled within the task graph as either edges or separate communication nodes. The nodes need to be designed to allow execution on one single resource at a whole without interaction to other resources.

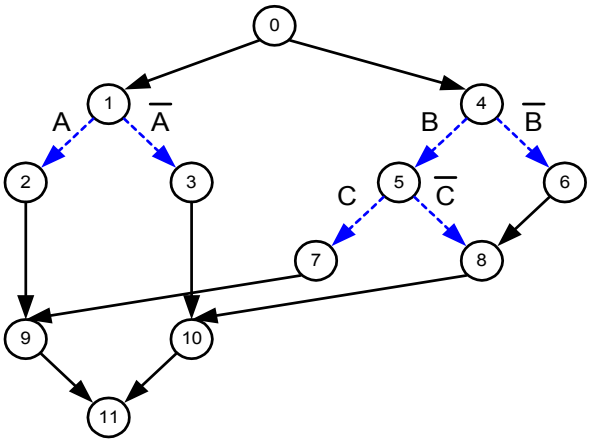


Figure 3.1: Example for a Conditional Process Graph (CPG)

The execution of control domain applications depend on the processed data. The control dependencies of the application are annotated as conditional branches within the CPG. Different paths through the CPG can be taken which are designated by conditional edges. Each node of the CPG can be the origin of conditional and unconditional edges. Figure 3.1 shows an example for a typical CPG. Node 1, 4, and 5 represent nodes with succeeding conditional branches depending on Condition A, B, and C respectively. Note that the non-determinism characterization of the design model does not yet select a specific branch, however, only one conditional branch can be selected exclusively for execution. This behavior can be compared to the *switch-case* instruction in the programming language C. The handling and processing of these conditions will be described in section 3.2.3 and section 3.3.3.

Eles et al., [33], introduce conditional process graphs as acyclic polar graphs $G(V, E_S, E_C)$. Each $P_i \in V$ represents one node. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \emptyset$ and $E_S \cup E_C = E$, where E is the set of all edges. An edge e_{ij} from P_i to P_j indicates that an output of P_i is an input of P_j. The graph is polar, which means that there are two nodes, called source and sink, that conventionally represent the first and last processes. These nodes are introduced as dummy processes, with no resource assigned, so that all other nodes in the graph are successors of the source and predecessors of the sink, respectively.

CPGs represent networking applications in a suitable way. Networking applications do usually not contain loops which cannot be unrolled to prevent the generation of acyclic graphs. Depending on the contents of the data packet, a specific path through the task graph is defined.

The worst case timing is the desired result of the analysis and determines whether the performance requirements are met. The synchronization of execution streams prevent a task from being executing unless all its incoming data transfers have been completed. Figure 3.2 compares the different situations for each condition. After node 0 has completed its execution, node 1 and 2 are started. Node 2 is done earlier than node 3. Nevertheless, node 4 waits until all incoming data has been received to derive the worst case scheduling.

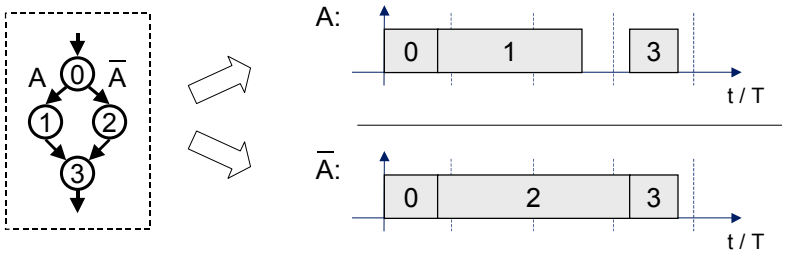


Figure 3.2: Simple Example for Scheduling of a CPG

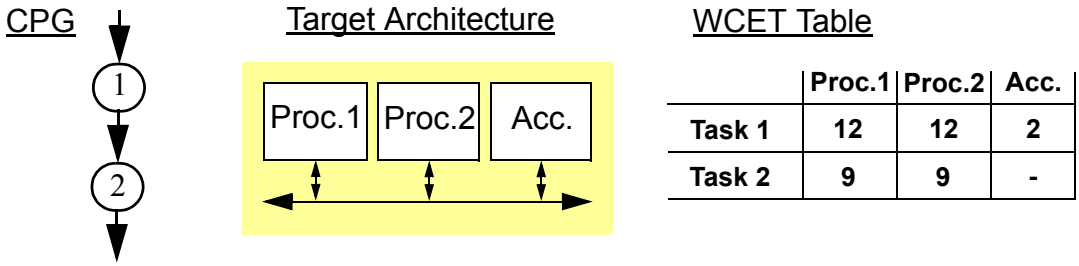
CPGs are useful representations of the applications and utilized in this thesis. Such CPGs may also be combiend to larger task graphs to represent the processing of several data packets. However, in this thesis only one instance of a CPG is used. In chapter 4, CPGs are used for evaluation which are composed of different smaller CPGs creating a larger task graph with specific characteristics.

Representation of the Resources

The implementation options of the single functionalities are itemized in the target architecture. Resources provided in the target architecture can be classified in computational or communicational resources. Computational resources are all kinds of microprocessors, application specific HW accelerators, FPGA etc., in short all types of data processing devices. Resources which support the communication on SoC are categorized as communicational resources. Bus structures or point-to-point connections are examples for interconnecting resources.

The computational resources are specified with the help of performance figures, so called worst case execution times (WCET). At this level of abstraction, it is a common way to characterize the performance of a resource by determining the upper bounds of the execution latency for each functionality. The magnitude of the WCETs discriminates the different types of computational resources. WCETs can be extracted by profiling executable high-level models with the help of associated profilers. Other related work has already covered this topic intensively, for instance, see Peters et al., [19], [78] and [79]. The estimated latency values represented by WCETs are normalized figures referencing to processing clock cycles or to a base time, such as µs or ns, and called time unit (T) in this thesis.

Figure 3.3 shows the combination of target architecture and WCET in a simple example. The given conditional process graph (CPG) provides the functionality which needs to be processed. The target architecture provides various implementations and corresponding resources for the processing of the tasks. The WCET table specifies the availability and the performance for each resource and task. In this way, further resources can be added easily to the analysis; or existing resources can be modified or erased. Here, task 2 cannot be executed on the HW accelerator. Each task can only be bound to one resource at a time. Different tasks can be executed in parallel on different resources as long as the sequence of the tasks is according to the CPG.



	Proc.1	Proc.2	Acc.
Task 1	12	12	2
Task 2	9	9	-

Figure 3.3: Target Architecture as Defined in the Specification

If two consecutive tasks are selected to be processed on different resources, context transfers of the processing units are necessary. A context transfer consists of suspending one process from execution on one resource, recording its current state, transfer this state to the other resource and starting the execution of the transferred process.

Representation of the Communication

Communication resources interconnect all computational resources of the target architecture. Bus structures or single connections in between two resources, both unidirectional and bidirectional, are the most common types of communication resources. Connections via multiple buses or multi-hop communication can also be integrated in the design model. However, these types of connections are not considered in this thesis.

I/O and Memory accesses are represented by latencies caused during the processing of the interfaces' buffer management, or the load/store operation of the memory controller. Since they have the same behavior during the analysis, I/O and memory can be assigned to the computational class of resources. The corresponding data transfers contribute to the occupancy of the shared communication resource.

For simplicity, only the total latency of the data transfer is used in the algorithms, comparable to WCET. At this abstraction level of the analysis model, the buffer sizes are considered as unlimited, since the scope of this work is not to determine these values, but to analyze the interaction within the entire architecture.

On each communication resource, only one data transfer can be performed at a given moment. If more than one resource wants to access the shared communication resource, an arbitration mechanism has to decide which resource' data transfer request is to be granted. Here, the resource which ask for access first, becomes granted for the transfer first.

The representations of the functionality, the resources, and the communication form the design model used in this thesis. The CPG and the WCET table contain all necessary information to perform partitioning.

3.2 Constructive Heuristic by Xie et al.

In chapter 2, the algorithm of Xie et al., [107], was identified as a very suitable partitioning algorithm that efficiently supports conditional branches in embedded systems. An ME detection algorithm is presented that helps the scheduling algorithm to exploit resource sharing. Binding and scheduling are performed simultaneously to take advantage of the resource sharing among those ME tasks.

3.2.1 Partitioning Algorithm

The partitioning algorithm consecutively performs the binding and the scheduling for each task of the CPG. The results may be evaluated within the architecture exploration loop. The following pseudo code in figure 3.4 gives an outline of the functionality of the scheduling algorithm introduced in section 2.2.5.

The calculation of the priorities for the list scheduling is performed in two steps. The result of the first step for each task is called static urgency (*SU*) which represents global priorities. Later on, this value is revised during scheduling and assigned as dynamic urgency (*DU*) which considers the local situation of the schedule.

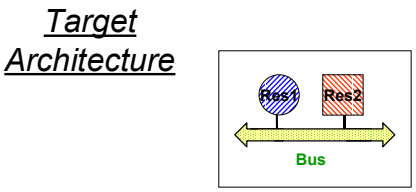
Static Urgency

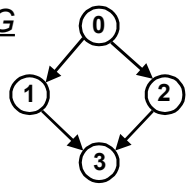
The static urgency (*SU*) is calculated for each node based on the maximum chronological distance of the considered nodes to the end node of the task graph, similar to

```
1.    for (task = each task of cpg)
2.        calculate static_urgency(task)
3.    create ready_list(list of ready task)
4.    while (ready_list is not empty)
5.    do
6.        if (any ready_task is partitioned on ASIC)
7.            schedule(ready_task; ASIC)
8.        else
9.            for (ready_task = each task of ready_list)
10.               for (pe = each CPU)
11.                   calculate dynamic_urgency(ready_task, pe)
12.                   determine max dynamic_urgency
13.                       with (ready_task_max; pe_max)
14.           schedule(ready_task_max; pe_max)
15.       update(ready_list)
16.   done
```

Figure 3.4: Outline of Partitioning Algorithm of Xie et al., [107]

priority assignment in some list scheduler. This figure can be compared to the HEFT (Heterogeneous Earliest Finish Time) algorithm, [93]. The HEFT algorithm selects the task with the highest priority at each step and assigns the selected task to a resource which minimizes its earliest finish time. Figure 3.5 shows an example for a calculation of the *SU*





Resource Library

WCET Table		
Task	Res1	Res2
0	**2**	-
1	**5**	**2**
2	**9**	**2**
3	**2**	-

Static Urgency Table

Task	SU	
0	**11.5**	= 8.5+1+ 2/1
1	**6.5**	= 2+1+ (5+2)/2
2	**8.5**	= 2+1+ (9+2)/2
3	**2**	= 2/1

Figure 3.5: Static Urgency (*SU*) Calculation

values. The weight for each task is calculated as the average WCET on available CPUs. Data transfers are considered as additional latencies with fixed duration of 1T. Each longest branch path is used to calculate the *SU* of each branch fork task.

The determination starts from the bottom of the CPG in a ALAP (as-late-as-possible) scheduling manner. Since the resource 2 is not available for node 3, the *SU* value of node 3

contains only the WCET of 2T. Nodes 2 is assigned an *SU* value of 8.5T consisting of the *SU* value of node 3, a data transfer of 1T, and the mean WCET of node 2. Similarly, node 1 results in an *SU* value of 6.5T composed of the *SU* value of node 3, a data transfer of 1T, and the mean WCET of node 1. For the calculation of the *SU* value of node 0, the latency of the critical path, here described by node 2, is used. The *SU* value of node 0 with 8.5T consists of the *SU* value of node 2, a data transfer of 1T, and the WCET of node 0.

Dynamic Urgency

With *SU* values determined before the scheduling, the dynamic urgency (*DU*) revises these priorities and takes actual scheduling and congestions into account. The *DU* is defined as, [107]:

$$\begin{aligned} DU(Task, CPU) = {} & SU(Task) \\ & - max\{t_{ready}(Task), t_{ready}(CPU)\} \\ & - WCET(Task, CPU) \end{aligned} \qquad (3.1)$$

The *DU* depends on the following factors:

- *SU* implies the chronological distance of the critical path to the last node of the CPG. High priority increases the chances of being preferred in the selection process. The larger the *SU* value, the more critical the node to be processed earlier than other nodes.
- $t_{ready}(CPU)$ is the earliest processing start time for the considered task on the considered CPU, e.g., cause by the earlier execution of preceding tasks.
- $t_{ready}(Task)$ is the finish time of the preceding task. This value takes into account the communication time of 1T for the data transfer from its predecessor.
- The worst case execution time (*WCET*) is used of the considered *Task* on the specific *CPU*.

The larger of the two t_{ready} values ensures that the data and the resources are available.

The highest *DU* value will be chosen to bind the corresponding task to the corresponding CPU. This *DU* value conforms with the dynamic priority in list schedule algorithm described in section 2.2.5.

3.2.2 Calculation of List Scheduling Priorities

To get a better understanding of the relevance of these addends of eq. (3.1), figure 3.6 should give an illustration of the various urgencies used with *ReCA*.

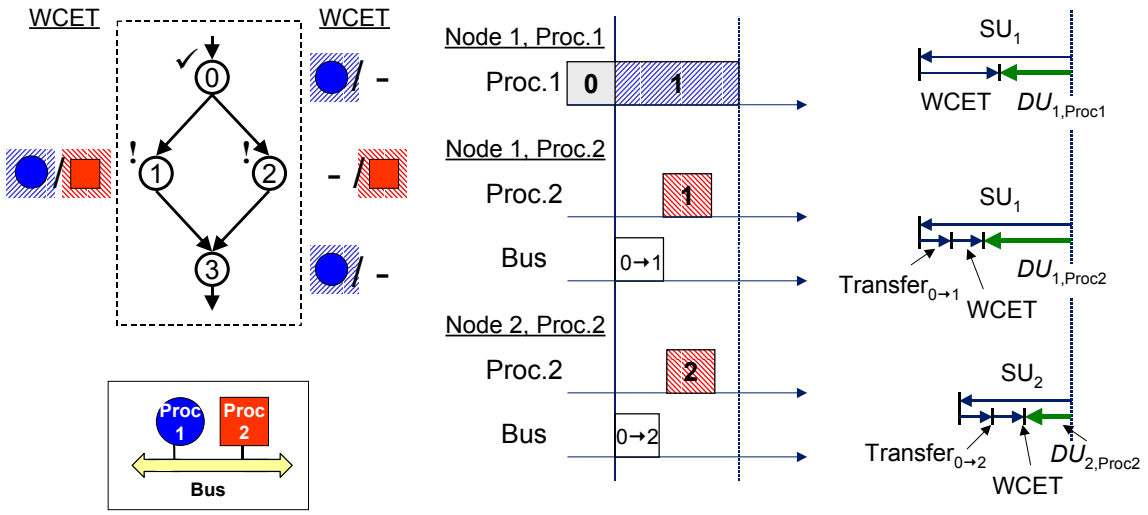


Figure 3.6: Scenario for Explanation of Urgencies

The left-hand side of the figure consists of a single CPG, a target architecture, and corresponding WCETs. In this example, node 0 is already scheduled. It is to be determined which ready tasks and which available resource are to be chosen next. Three node/resource combinations are available:

- Node 1 can be processed on the microprocessor 1 or 2 ("Proc.1" or "Proc.2").
- Node 2 needs to utilize the microprocessor 2 ("Proc.2").

These three scheduling possibilities are shown in the middle of figure 3.6. Each schedule expresses the allocation of the involved resources. The right-hand side of figure 3.6 illustrates the calculation of *DU* with the help of vectors. As the first potential solution, node 1 would follow task 0 on processor 1. This situation does not require any data transfer. *SU* represents the predetermined global priority and WCET the actual processing duration. For the other two solutions with the processor 2 as selected resource, bus activity is necessary since data needs to be transferred between the processors.

As a result, the $DU_{1,\mathrm{Proc.2}}$ vector has the greatest length which corresponds to the earliest finish time of the three possibilities. In addition, this vector length represents the greatest time buffer for the remaining tasks to be executed. Although the WCETs for node 1 differ in a factor of 2.5., the length of the two resulting *DU* values of node 1 are similar. Data transfers can cause an unexpected high increase in latency. The length of the transfer vector

is automatically adjusted to the allocation of the involved resources regardless of the type of resources.

3.2.3 Detection of Conditional Branches

Xie et al. introduces a branch labeling method to identify conditional branches in the CPG. The available conditions are specially selected to ensure that only one conditional branch per node can be chosen at a time. In this way, the conditional branches mutually exclude each other for execution. In this way, the handling of conditional branches is facilitated within task graphs.

Each node in the CPG is associated with a branch information structure as follows:

Level is the number of branch fork tasks that have to be executed before reaching this tasks. **Branch_label[i]** is the name of the i^{th} level branch fork task. **Branch_cond[i]** is the condition value for the i^{th} level branch.

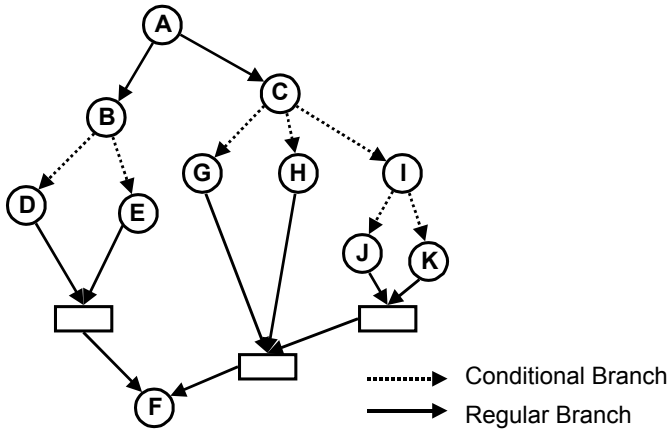


Figure 3.7: An Example Condititional Process Graph

Figure 3.7 depicts an example of the conditional process graph used by [107]. The following branch-labeling recursion algorithm outline is applied and the results of figure 3.7 are presented in table 3.1.

Node i	Level	Branch_label	Branch_cond
A	0	N/A	N/A
B	0	N/A	N/A
C	0	N/A	N/A
D	1	B	B1
E	1	B	B2

Table 3.1: branch_info struct for Figure 3.7

Node i	Level	Branch_label	Branch_cond
G	1	C	C1
H	1	C	C2
I	1	C	C3
J	2	C, I	C3, I1
K	2	C, I	C3, I2
F	0	N/A	N/A

Table 3.1: branch_info struct for Figure 3.7

Mutual exclusiveness (ME) of two any task can be identified by the algorithm in figure 3.8.

```
1.    function ME(task1, task2)
1.    {
2.        L = min(task1.level, task2.level)
3.        if (L = 0) then mutual_exlusive = false
4.                        // not ME such as task A and task I
5.        if (L > 0) then
6.            for all L levels (i)
7.                if (task1.branch_label[i] ≠ task2.branch_label[i])
8.                    then mutual_exlusive) = false
9.                        // not ME such as task I and task E
10.               if (task1.branch_label[i] = task2.branch label[i] &
11.                   task1.branch_cond[i] ≠ task2.branch_cond[i])
12.                   then mutual_exlusive = true
13.                       // ME such as task H and task K
14.               else if (i+1 > L)
15.                   then mutual_exlusive = false
16.                       // not ME
17.           end
18.       return mutual_exlusive
19.   }
```

Figure 3.8: Outline of Mutual Exclusiveness Detection of Xie et al., [107]

ME communication edges can be determined by using this scheme. For instance, the communication edges B-D and B-E in figure 3.7 are ME and can be allocated to the same communication link having overlapping execution time slots in the scheduling. Also, note that two tasks on different branches might not be ME, for example node E and G.

The ME is reflected in the CPU ready time $t_{ready}(CPU)$ for any partitioned task. The algorithm determines $t_{ready}(CPU)$ considering ME is shown in figure 3.8.

```
1.    function PE_available_time(Task ready_task, CPU pe)
2.    {
3.        if (no task scheduled on pe)
4.            return t_ready(pe) = 0
5.        set sched_task = latest allocated task on pe
6.        while ( ME(ready_task, sched_task) )
7.        do
8.            sched_task = sched_task.previous_task
9.        end
10.       return t_ready(pe) = sched_task.completion_time
11.   }
```

Figure 3.9: Calculation of CPU Ready Time

3.2.4 Examples

Figure 3.10 depicts an example of a schedule of the CPG in figure 3.7 taken from [107]. The target architecture consists of two CPUs and a shared bus, which is not shown in the schedule. Tasks G, H, and J are determined as ME. Hence, these three tasks may have overlapping execution time slots on CPU 1. Similarly, task D and E are ME and can also be scheduled at the same time slot. E and K belong to different conditional branches and therefore are not ME. Task K needs to wait until E is processed.

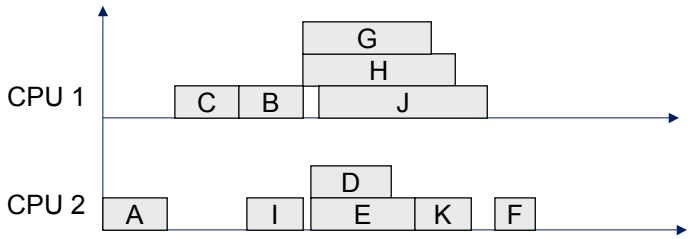


Figure 3.10: Scheduling Result for Example in Figure 3.7

The objective of this algorithm is to determine the worst case scheduling, which can be identified as the task sequence A-C-I-J-F. This path has the longest execution duration. This knowledge is very important during co-synthesis, which has to find out the architecture that accommodates all cases.

In the next section, the Reference Constructive Algorithm (*ReCA*) is introduced based on the algorithm of Xie et al. and serves for further evaluations.

3.3 Reference Constructive Algorithm

The partitioning methodology of Xie et al. provides features necessary for the partitioning of design model required in this thesis. However, this algorithm is mainly focused on multi-processor systems. For the support of heterogeneous SoC architectures, some modifications to the algorithm of section 3.2 need to be carried out. Two major changes are the enhancement of ME detection and the treatment of components within the target architecture. These modification are necessary to allow the evaluation of performance with an ample number of synthetic task graphs. The modified partitioning algorithm is called Reference Constructive Algorithm (*ReCA*) and used as reference in the remainder of this thesis.

3.3.1 Algorithm Adjustments

The following enhancements are included in the *ReCA*:

- No hardware preference

  The algorithm of Xie et al. focus on multi-processor systems and privilege HW accelerators. *ReCA* does not differ between software and hardware resources as computational units. There is no preference for non-CPU blocks. HW accelerator blocks are treated as high-performance application specific processors.

- More flexible Communication Structure

  ReCA supports a flexible communication structure between computational resources of the target architecture. An arbitrary number of busses can be established between the resources. Also, it can be differentiated whether the resources are allowed to write or read on several buses. In this way, unidirectional connections can be created in the design model.

- Considering bus allocation and congestions of variable length data transfers

  For multi-processor analysis, the duration of the transfers between the units can become significant compared to the execution duration of the tasks. Instead of simply adding one time unit to consider context transfers, an estimation of time units depending on the amount of data is used for the calculation of bus allocation. This value is specific to the edges of the CPG.

- All resources are considered in *SU* calculation.

  The algorithm of Xie et al. only regards CPUs for the *SU* calculation. *ReCA* considers all computational resource, since they also contribute to the total latency of the application.

- Hole filling of empty slots in the entire schedule

  To simplify the scheduling process of the algorithm of Xie et al., only the latest scheduled task of the selected resource is considered to find the next available time slot. *ReCA* removes this restriction. The entire time frame of the schedule is examined to find the earliest possible slot fitting the processed task. Figure 3.11 show an example of a task graph whose node 3 and 4 have no data dependency. Since node 2 is executed later than node 1, *ReCA* can schedule node 4 sooner than the algorithm of Xie et al.. In this way, a compacting of the scheduling results in the improvement of performance by utilizing unassigned slots in an earlier point of time in the schedule.

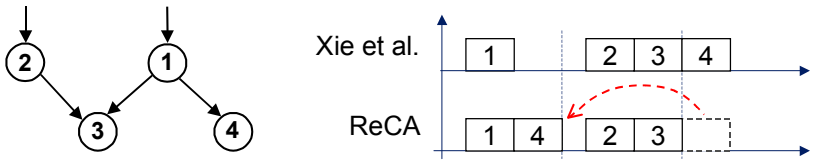


Figure 3.11: Hole Filling of Empty Slots

- Enhanced ME Detection for Arbitrary Connections

  ReCA supports arbitrary communication structures within the model introduced by Wild in [101]. ME can be detected without having a regular structure of the graph which facilitates the generation of analysis models. Graph generation tools, such as TGFF [24], can be applied without considering the graph's structure for conditional branches.

3.3.2 Implementation

According to figure 3.4, the following pseudo code of the partitioning algorithm, figure 3.12, gives an outline of the functionality of the *ReCA*, and figure 3.13, an outline of the updated static urgency calculation.

```
1.    calculate Static_Urgency for each node using all resources;
2.    while(nodes are left to be mapped)
3.    do
4.        determine all ready_nodes
5.        for (N = all ready_nodes)
6.            for (R = all possible resources)
7.                Dynamic_Urgency(N, R) = Static_Urgency(N)
8.                    - max(t_ready_data(N); t_ready_res(R))
9.                    - WCET(N, R)
10.               (Nmax, Rmax) = determine max Dynamic_Urgency(N, R)
11.           end
12.       end
13.       bind_and_schedule(Nmax,Rmax);
14.   done
```

Figure 3.12: Outline of *ReCA*

```
1.    while (not all nodes are processed)
2.    do
3.        for all nodes(node)
4.            if (node == last node)
5.                SU(node) = average_WCET(node)
6.            if (all successors are processed)
7.                for all successors(sux)
8.                    SU'(sux) = SU(sux) + communication_latency
9.                    determine max(SU'(sux))
10.               end
11.               set SU(node) = max(SU'(sux)) + average_WCET(node)
12.               mark node as processed
13.       end
14.   done
```

Figure 3.13: Outline of *SU* Calculation

3.3.3 Improved Condition Support

Conditional branches are a principal characteristic in control-driven applications. For the facilitation of the CPG processing, conditional branches are usually handled each as single contiguous subsets of nodes without any interconnects to each other, such as in [107]. Connections between these contiguous subsets are not allowed in this methodology, see figure 3.14.

This constraint of the condition handling restricts the flexibility of the creation of design models. Arbitrary interconnects are necessary, e.g., for error handling or premature program termination require a break out of these single contiguous graph branches. Instead of creating a special error handling for each graph branch, a common subset of tasks for

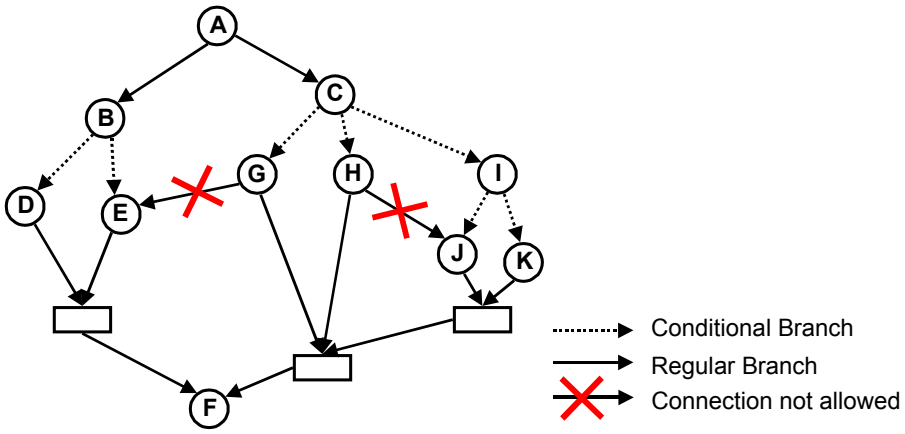


Figure 3.14: Traditional Conditional Branches

error handling is used by the application to achieve more efficiency. To support such combination, changes in the handling of conditional branches need to be made.

Figure 3.15 depicts an other example of a conditional task graph. All conditional branches are marked with a dotted line. Obviously, node 2 and 3 are ME because they are derived from the same conditional node and cannot be reached by other nodes. Any two nodes which originate from a common conditional node can be potentially designated as ME. Similarly, node 7 and 8 are ME since both originate from node 5. Although node 8 can additionally also be reached via node 6, this does not influence the ME of node 7 and 8. Node 7 does not have any direct relationship or connection with node 6.

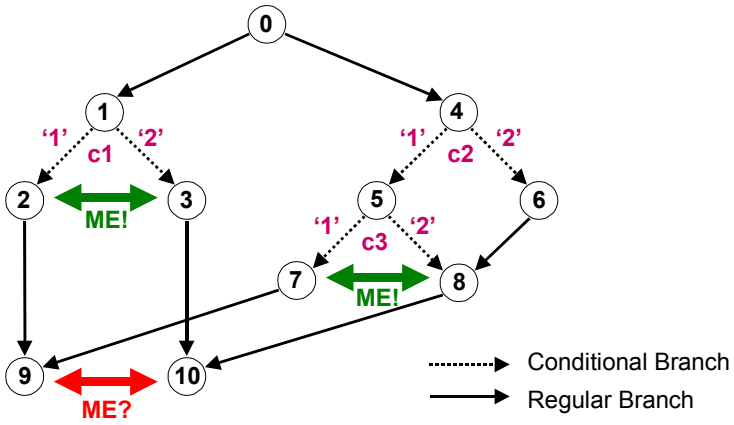


Figure 3.15: Mutual Exclusion of Tasks

Whenever it is possible to reach two different nodes under a common condition combination, these two nodes are not ME. For instance, node 9 and 10 cannot be identified by inspection. To detect ME, all paths to these considered nodes in the task graph have to be analyzed whether they allow overlapping allocation. A two step approach with an annotation stage and an inspection stage enables this analysis, first introduced by [7] and [101]. This improved condition support is used by *ReCA*.

3.4 Performance Evaluation of the Reference Algorithm

For the evaluation of the partitioning algorithms, a huge number of different task graphs need to be provided to allow statements of the performance. A manual creation of task graphs in a sufficient quantity is not feasible. Statements about the quality of the algorithms which are statistically ensured cannot be provided. Moreover, task graphs are specified with a large number of parameters. A limited number of manually generated task graphs based on actual applications cannot cover all parameter in a sufficient way. Only restricted statements about the partitioning algorithms can be made.

Hence, an approach is necessary to generate an ample number of synthetic task graphs in a fast and reproducible way. Such sets of task graphs needs to be adequately distinguishable to establish various classes of graphs properties. These classes of graphs properties are used to evaluate different behaviors of the algorithms. In this way, the performance can be predicted based on the respective parameters.

The synthetic design models represent a substitution of real-world applications which may serve as benchmark for evaluation of the algorithms. Also, the characteristics of the target architecture needs to be parameterized to substantiate the tests with different scenarios in order to assess the performance. A complete inspection of all parameter combinations is not feasible due to complexity. In a later chapter, a design model of a real-world scenario with various target architectures is used to confirm the derived trends.

In the following, the task graph generation tool "Task Graph for Free" (TGFF), [24], is introduced. The utilized parameters are used for the evaluation of the partitioning algorithms. The evaluation environment and tools are introduced which uses the design model and architecture assumption - the characterization of the graphs and the target architecture for the evaluation. The results of the performance evaluation conclude this section.

3.4.1 Generation of synthetic test pattern

The tool "Task Graph for Free" (TGFF) which was developed at Princeton University, [24], is applied for the generation of CPGs. These CPGs are used as test scenario references for various methodologies in the system synthesis research field. TGFF meets the requirements mentioned above and facilitates the generation of pseudo-random CPGs and parameters which specify the target architecture and the therein deployed blocks. Figure 3.16 shows an example of an task graph generated with TGFF.

For the analysis of the partitioning algorithms, parameters of synthetic graphs need to be designated. Due to the heterogeneity as well as the different levels of abstraction, the CPGs of actual applications cannot simply be defined by a limited number of parameters. The usage of such synthetic graphs is simply an aid for performance evaluation. With the possibility of a multitude of possible tests and a variety of parameter values, the indication of the performance of the considered algorithms may derive trends of their behavior.

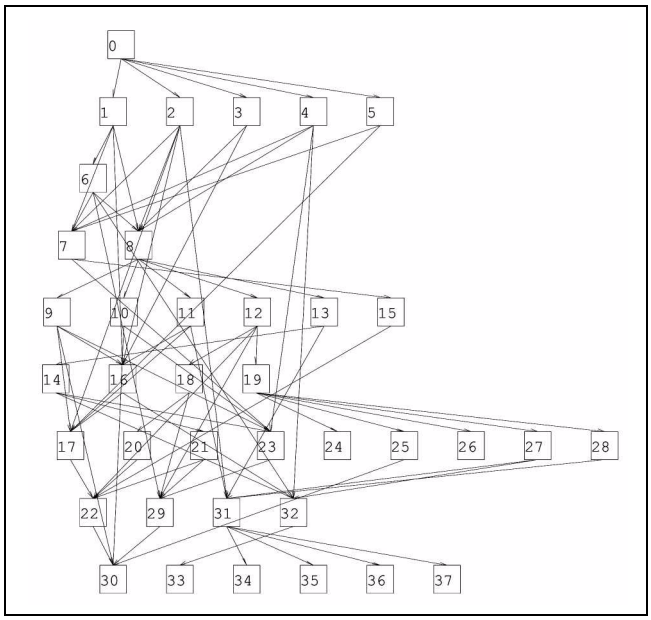


Figure 3.16: Example of a Task Graph Generated with TGFF

The following research groups use TGFF for the evaluation of their algorithm, just to name a few: Jeong et al., [54], Schmitz et al., [82], Shin et al., [84], Vallerio et al., [94], Xie et al., [107], Zhang et al., [108], etc.

Characteristics of the CPG

For the characterization of the synthetic graphs, the following properties are considered predominantly:

- Number of nodes
- Connectivity of the nodes and structure of the graph
- Amount of data transferred between nodes
- Number of conditional edges

For the reproduction of task graphs, these properties are suitable parameters to distinguish between different classes of task graphs. The structure of the graph is an essential property. However, it is difficult to specify the properties in numerical figures. Possible candidates for parameters can be the number of start and end nodes of a task graph, and the mean number of preceding and succeeding edges per node. In consideration of the considered application of network packet processing, it is expedient that the graph consists of one single start node and one single end node. The single start node represents the arrival of an data packet or the provision of the data packet header. The single end node demonstrates the storage of the information how to deal with the packet, or the forwarding

of the packet itself. Areas in the CPG with a different density of edges per node are not possible due to the input of the graph generator.

Characteristics of the Target Architecture

Besides the structure of the CPG, the target architecture is the other important input for the partitioning algorithms. A simultaneous generation of target architecture information containing performance figures is preferable, because this information is tightly linked to the nodes in the graph. The following parameters can be identified:

- Structure of the architecture
- Communication architecture
- Number of blocks/block types (resources)
- WCET of the tasks depending on the resources
- Implementation possibility of tasks on the resources

ReCA is able to process multiple buses of the target architecture. However for the evaluation in this thesis, the communication architecture consists of only one shared bus connecting all resources. Hence, the characteristics of the target architecture depends merely on the number of diverse resources and the WCETs depending on the task and resources. Furthermore, the implementation possibility decides about the feasibility of tasks which can be executed on a specific resource. Microprocessors can usually process all kinds of tasks, possibly with performance drawbacks, while application-specific accelerators can only execute specific nodes of the CPG.

Generation of synthetic graph and architecture pattern

The main parameters of the graph generation with TGFF are the number of nodes and the specification of the connectivity. With the given maximum number of preceding and succeeding edges per node, the graphs are generated by adding further nodes at the ingress and egress side of a first kickoff node recursively. Beginning with this first node, nodes are appended to these nodes to the ingress and egress side until the desired number of nodes is reached. For instance, on the ingress or egress side, the node with the largest difference x of the maximum allowed and actual number of connections on the corresponding side of the node is selected. A random number y of nodes with $y \in [0;x]$ are appended on this node connected by edges. In this way, the structures of the graphs are created by nodes and edges.

In addition, parameter values can be annotated to the nodes of the graph and are represented by so-called tables. They are selected in a pseudo-random manner similar to the graph generation. The application and interpretation of the parameters is left to the user of the tool. Parameters of the graph and the target architecture is defined by both pre-set and statistical values which are uniformly distributed and specified by the mean and the half interval width. The relevant parameters for the analysis are primarily the amount of data for

communication, as well as the feasibility and the WCET of the tasks subject to the resources. Further information can be found in the TGFF documentation in [24].

For the generation of test data CPGs, following parameters are used:

- Number of Nodes (n)
- Number of maximum preceding (id) and succeeding (od) edges of the nodes
- Amount of data and corresponding latencies for context transfers.

The actual number of nodes within a CPG is not exactly n, but within the range of $[n;n+od-1]$ due to the way of implementation. Since od is quite small compared to n, the deviation is negligible. Therefore, a specific value of the graph size should be understood as a class size.

The following parameters characterize the component's properties of the target architecture used by TGFF:

- Number of resources per resource types
- Type of resources
- Corresponding WCET of each task with regard to each resource.

Previous work analyzes the characteristics of computational resources within SoC architectures. Of special interest is the performance and the distribution of instruction types. CommBench, a benchmark for evaluating and designing telecommunications network processors introduced by Wolf et al. in [104]. The benchmark applications focus on small, computationally intense program kernels typical for SoC with network processors. Resource properties can be assigned to the distribution given in [104]. This would allow to model these pseudo applications and to evaluate the partitioning algorithms under proper scenarios. Later in this thesis, the design models are generated according to CommBench.

3.4.2 Evaluation Environment and Tools

The simulation environment consists of a tool chain generating and modifying task graphs with potential target architectures. Starting with a plain generic task graph, information about conditional branches are included, and memory accesses are inserted in the CPG. The prepared graphs are fed in the partitioning algorithm. Finally, the results are compiled and presented.

Figure 3.17 gives an overview of the tool chain for the following analyses:

- *tgff*: Task Graphs For Free (TGFF), [24], generates pseudo-random task-graphs and WCET figures for the target architectures.

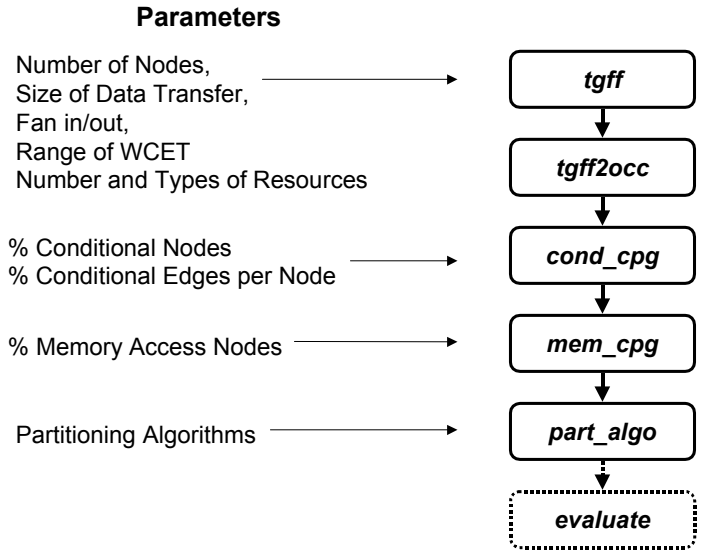


Figure 3.17: Tool Chain for the Analyses of the Partitioning Algorithms

- *tgff2occ*: This graph format conversion tool converts the output of TGFF to another format used by the constructive algorithm tool *eca*.
  Furthermore, TGFF cannot sufficiently distinguish the computational resources with regard to the implementation of the processes. Processors can process any tasks performance-wise to a greater or lesser extent, whereas application-specific resources are designed to support only a few task of the task graph, but enable high-performance processing. For that reason, the ratio of processible tasks for each resource can be specified during graph generation by the tool *occ2cpg*. In this way, the feasibility of the application regarding the resources can be represented.
- *cond_cpg*: With the help of this tool, branches of the graph can be assigned conditional. Additionally to the percentage of total nodes becoming conditional, it can be defined how many percent of the branches of a selected node will be conditional.
  TGFF is missing the feature to describe conditional branches within the generated task graph. Therefore, a task graph modification tool designates a percentage of nodes as sources for conditional branches. The number of ME conditional branches or edges starting from these nodes can be set as well. The selection of these nodes and branches is made randomly.

- *mem_cpg*: This tool appends memory nodes in the existing task graph. For that purpose, designated links are split to insert a memory access node. In addition, the memory is append to the target architecture as a further resource. In this way, memory access can be modelled without major modifications of the CPG.
- *part_algo*: This tool represent the implementation of the different partitioning algorithms and *ReCA*.
- *evaluate*: The results of the algorithms will be summarized. The output figures are the average improvement of the evaluated partitioning algorithm compared to *ReCA* in percent, the ratio of improvements, and the ratio of deteriorations of performance.

In the appendix B, the command options of the used tools are described.

3.4.3 Evaluated Partitioning Algorithms

For the performance evaluation of *ReCA*, other partitioning algorithms are necessary which support conditional branches in their design model. In addition, the algorithms need to process synthetically generated design models in a sufficient quantity to allow statements about the performance.

The subsequent sections are following the partitioning algorithm evaluation of Wild, [100] and [101], since the evaluated algorithms support conditional branches. Four iterative heuristics with tabu search are compared against the reference algorithm, *ReCA*, introduced in section 3.3:

- *ReCA*: The Reference Constructive Algorithm represents a constructive greed heuristic algorithm based on the algorithm described in section 3.3.
- *TS-One*: Tabu Search (TS) ([39], [40]) which is already introduced in section 2.2.2 searches the neighborhood of the current solution in the solution space. To avoid oscillating effects between two solutions, a tabu list includes all selected solutions and rejects already made decisions. With a predefined number of iterations without improvement of the solution, the search is terminated. Also, the exhaustion of all solution possibilities in respect of the tabu list does not leave further solutions and results in termination of the algorithm. This variant *TS-On*e produces new solutions which differentiate in one change of the binding; therefore the name *TS-One*. In this way, the solution space can be analyzed seamlessly, however less directed.

- *TS-CP*: The critical path (CP) variant of tabu search uses the tasks in the critical path of the CPG to choose changes of the mapping. The CP illustrates the longest execution path within the CPG and determines the overall latency of the application. Hence, these tasks are very promising for changes in mapping to improve the performance of the application.
- *TS-ECP*: In addition, the enhanced critical path (ECP) variant utilizes both computational and communicational nodes of the critical path. In addition, non-CP data transfers preceding data transfers of the critical path, computational nodes of these non-CP data transfers, and computational tasks preceding nodes of the critical path having already a delay. These delays are so-called wait-times. The main objective of *TS-ECP* is to reduce these wait-times.
- *FAST*: The "Fast Assignment using Search Technique" (*FAST*) algorithm is introduced in [62] and revised in [64]. It acts as an alternative iterative heuristic algorithm. The search algorithm is carried out in two nested loops searching the critical path. The maximum number of iterations is limited by two fixed parameters. The inner loop is specified by a margin parameter and aborted early when no better solution is found.

3.4.4 Design Model and Architecture Assumptions

The following parameters describe classes of different design model and target architectures. These classes of design model facilitate the evaluation of the partitioning algorithms. The behavior and characteristic of the algorithms can be rated on the basis of this design model classes.

The design model consists of 100 nodes in the average. Each node has 5 preceding nodes and 5 succeeding nodes in the mean. Regarding conditional branches, 10% of the nodes within the CPG are designated conditional. Out of these conditional nodes, 80% of the succeeding edges are assigned as conditional branches.

The target architecture contains two type of resources. Resource type A represents resources which can process all kinds of tasks, such as microprocessors. All instances of resource type A are identical. Resource type B process only certain selected tasks of the application (here 25%) with a mean speed-up factor of 5 faster than resource type A. These selected tasks are arbitrarily chosen. Such resources may represent application-specific accelerators. Each instance of resource type B represent a individual resource which has its own characteristic. The computational resources are connected by a common shared bus. The mean amount of transferred data accounts for 50 bits, if not overridden by the analysis boundary conditions.

The following table 3.2 represents a typical set parameter for potential scenarios used for the analysis later in section. The following constraints are applied to the tests as long as no other constraints are given:

Parameter	Value Range
Design Model	
Size of Graph	100
Number of Preceding Branches per Node	5
Number of Succeeding Branches per Node	5
Ratio of Conditional Nodes	10 %
Ratio of Conditional Edges originate from a Conditional Node	80 %
Target Architecture	
Number of Resources on Resource Type A (homogenous)	2
Number of Resources on Resource Type B (heterogeneous)	5
Speedup of Resource Type B compared to Resource Type A	5:1
Ratio of executable Nodes on Resource Type A	100 %
Ratio of executable Nodes on Resource Type B	25 %
Number of Buses	1
Average Data over Bus	50 bits

Table 3.2: Simulation Properties of CPGs and Target Architectures

3.4.5 Results

In the following, the results of the evaluation are shown. With the help of TGFF, the various algorithms are analyzed. The parameters described in section 3.4.1 are varied and the impact is investigated. Since these parameters are describing too many potential degrees of freedom of the application model to simulate and display completely, only a limited range of values, each varied one at time, are examined.

The following result figures depict no absolute schedule latencies due to the synthetical character of the graphs. Only the comparison of different algorithms is interesting. Hence, all values are in relation to performance of the *ReCA* algorithm. To conserve the facility of inspection, the figures simply consist of the mean of 60 different graphs. The calculations are performed on an AMD Athlon XP 1700+ and Linux compiled with *gcc* and the optimization option *'O3'*. The execution durations are specified as absolute time values to get the impression of calculation effort across.

Design Model Properties

In figure 3.18, the influence of the CPG size is depicted. Each working point is determined by *n* results of the considered partitioning algorithm and set in to relation with *ReCA*:

$$Reduction\ in\ Latency = \frac{1}{n}\sum_{N}\left(\frac{Latency(Algorithm) - Latency(ReCA)}{Latency(ReCA)}\right) \qquad (3.2)$$

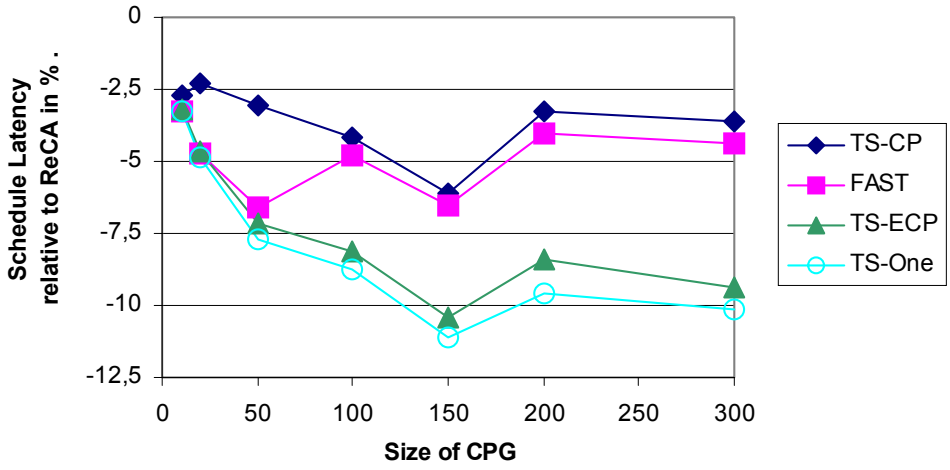


Figure 3.18: Schedule Latency Depending on the Size of the CPG Relative to *ReCA*

The fixed number of cycles within the search loop limits the achievable performance of the *FAST* algorithm with an increasing CPG size. Tabu Search (*TS-One*) can perform better, since the abort criterion is a complete neighborhood search and this results in a better coverage of the solution space. Tabu Search with a critical path neighborhood (*TS-CP*) performs about the same as the *FAST* algorithm. *TS-CP* is using nodes from the critical path. Therefore, the solution space and the achieved performance are limited. With an enhanced neighborhood and an increase of number of iterations, the *TS-ECP* can outperform *TS-CP*. Since only a part of all possibilities are considered, the results of *TS-One* cannot be reached. The degraded performance of the constructive algorithm (*ReCA*) compared to the iterative algorithms lies in the simultaneous decisions of binding and scheduling based on assumptions of the potential future scheduling which is not known yet. Made decisions cannot be changed later on, and no rescheduling is possible. Iterative algorithms can revise the binding step by step considering the entire scheduling. Hence, all iterative partitioning algorithms perform better than *ReCA*.

In this context of the functionality of the algorithms, the consequences in respect of runtime of the algorithms are shown in table 3.3. For the iterative algorithms besides on the size of CPG, the runtimes particularly depend on the number of examined alternative mappings. The trend of the size of neighborhood correspond to the figures in table 3.3. The

analyzed mapping possibilities of *TS-One* and *TS-ECP* increase significantly, so do the runtimes. The short runtimes of *TS-CP* compared to *TS-ECP* base upon the smaller solution space.

The use of iterative algorithms based on local search is heavily limited due to the strong correlation of graph size and runtime. The advantages in performance of the TS algorithms, shown in figure 3.18, compared to *FAST* and particularly *ReCA*, which is able to process much bigger CPGs due to shorter runtimes, are highly counterbalanced.

Size of CPG	**Runtime**				
	ReCA	***TS-One***	***TS-CP***	***TS-ECP***	***FAST***
50	0.01s	1.40s	0.36s	0.58s	1.71s
100	0.04s	13.43s	2.17s	5.84s	6.66s
150	0.13s	5.865s	6.16s	25.63s	17.17s
200	0.31s	187.71s	14.37s	81.87s	37.26s
300	1.20s	1198.45s	55.44s	810.97s	128.80s

Table 3.3: Runtime of the Algorithms with Regard to the CPG size

Architecture Properties

The following investigations have their emphasis on the number of processors and accelerators of the target architecture. In figure 3.19, the simulation results for all the presented algorithms with *ReCA* as reference are depicted. The axes show the numbers of processors and accelerators and the mean scheduling latency in relation to *ReCA*.

Also for this analysis, *TS-One* and *TS-ECP* behave similarly. With an increasing number of computational resources, the performance advances due to the increasing number of alternative mappings by the increasing size neighborhood. On the other hand, the *FAST* algorithm takes less mapping alternatives into account, the more resources are utilized. The increasing solution space is processed more and more fragmentary. For *TS-CP*, no consistent trends can be observed. Better performance can only be achieved with a high number of processors and a low number of accelerators. An increase in the number of accelerators, results in a higher bus usage and therefore worse performance. *ReCA* performs about the same as the *FAST* algorithm with a low number of accelerators. An increasing number of accelerators here, lowers the performance, since early decision cannot be revised later on in the scheduling and causes unfavorable situations. In particular, the processed data on the accelerators which needs to be sent back to the source resource is not regarded sufficiently. Overall, the performance level of the constructive algorithm is worse than the level of iterative algorithms.

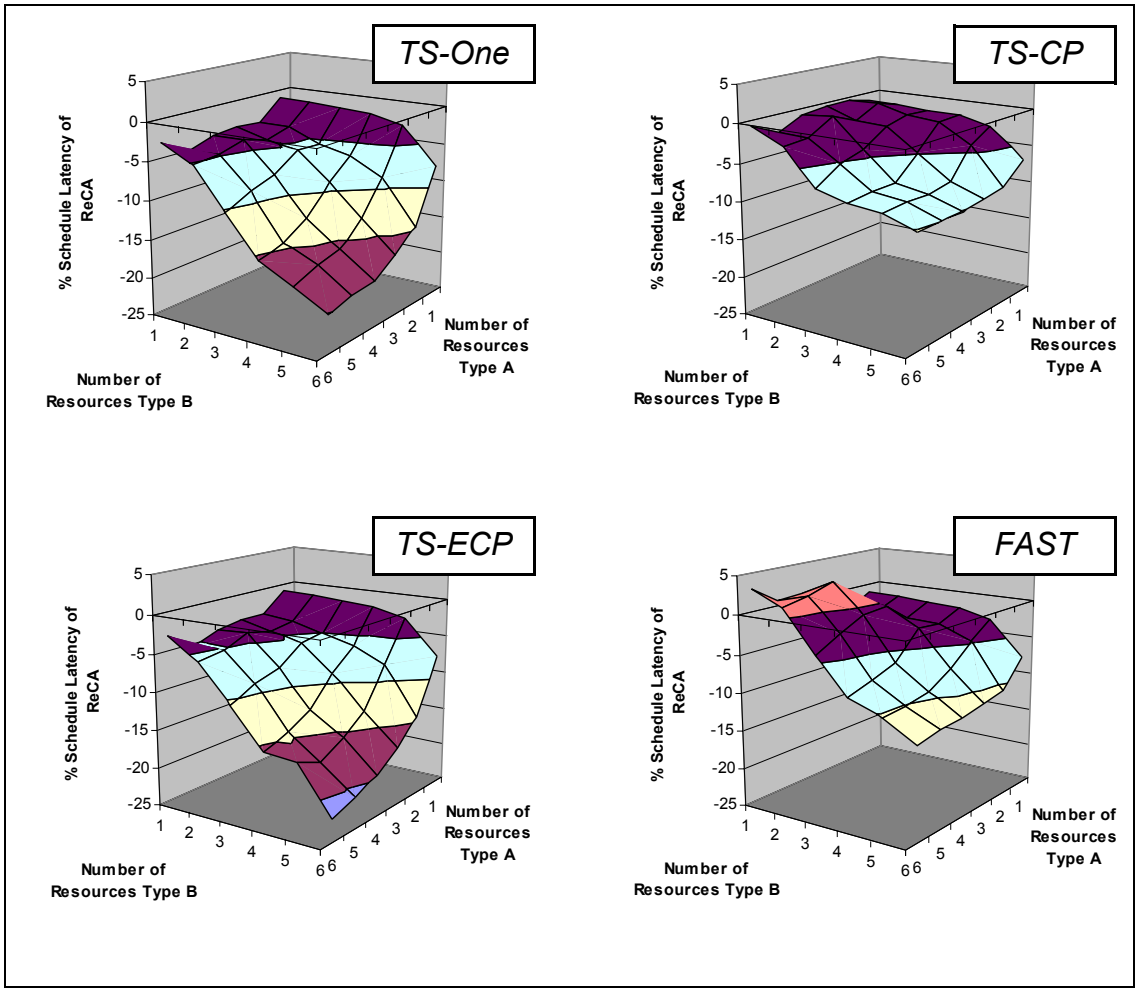


Figure 3.19: Schedule Latency of Different Partitioning Algorithms Depending on the Target Architecture Relative to *ReCA*

3.5 Summary and Conclusions

Based on the constructive greedy algorithm by Xie et al., [107], the Reference Constructibe Algorithm (*ReCA*) is introduced in this chapter. The major modifications compared to [107] are:

- No differentiation is made on types of computational resources.
- The bus allocation and congestions of variable length data transfers are considered.

- Empty slots are filled in the entire schedule during partitioning.
- All resources are considered in static urgency (*SU*) calculation for the global priorities of the list schedule.
- A more flexible communication structure is allowed.
- Enhanced Mutual Exclusiveness (ME) Detection for Arbitrary Connections replaces the existing method.

To assess the impact of these modifications, an evaluation setup with the task graph generation tool TGFF is arranged. The analysis results show that the run-time of *ReCA* is superior compared to the iterative heuristics. The larger the size of the CPG, the longer the runtimes of iterative algorithms, even in orders of magnitude. The high speed of *ReCA* is confirming the selection of a constructive algorithm for partitioning.

Nevertheless, the performance of *ReCA* is in need of improvement. The iterative partitioning algorithms produce up to 23% shorter schedule latencies in the mean in this scenario. The following chapter introduces enhancements of *ReCA* to augment the quality of binding and scheduling choices and improve the performance. In this regard, supplementary information is utilized during the determination of the priorities.

Chapter 4

Enhanced Constructive Algorithm

The advantage of constructive algorithms delivering a solution in a shorter analysis time than recursive algorithms is shadowed by impaired results shown in chapter 3. To improve the performance, *ReCA* is modified to include additional information in priority calculation.

Two factors are accountable influencing the quality of the results: the resource selection and the sequencing of tasks. By using more scheduling information of future processing steps of the application during partitioning, the outcome of this algorithm can be improved. This algorithm is called Enhanced Constructive Algorithms (*ECA*) in the following.

Analysis models with different levels of granularity can cause constructive algorithms to generate forbidden results. Although two consecutive task are meant to be bound to the same resource, the algorithm can utilize different resources during binding. The preselecting of binding possibilities can solve such issues. Nodes having similar binding constraints can be pooled to clusters of Common Implementation Nodes (CINs). In this way, different resource sets composed of resources of the target architecture allow the consideration of the interplay of various different resource types. Comparing different sets of resources is a novel approach, since other partitioning algorithms allow only the comparison of single resources. By using various sets of clusters for the same task graph, different design objectives can be pursued during partitioning. Design objectives can place special emphasis on the outcome of the analysis, such as parallel processing or the acceleration of certain functionalities.

In the following, the enhancements of *ECA* with Look-Ahead (*LA*) are introduced, applied and analyzed. The analysis environment of chapter 3 is utilized for the analysis of *ECA*. Subsequently, the third enhancement - clustering - is introduced and analyzed. The implementation is evaluated in a modified environment using compound task graphs

4.1 Partitioning Issues and Motivation for Improvement

In the previous chapter, the comparison of the partitioning algorithms shows that the constructive approach is faster in processing partitioning compared to iterative algorithms. The reason for this execution time advantage is the fact that each partitioned task cannot be assigned to other resources or rescheduled. In this way, more favorable binding and scheduling possibilities may not be achievable. The outcome confirms that it is necessary to improve the quality of the results.

The two main components of partitioning are binding and scheduling, as introduced in section 1.2. Constructive algorithms intertwine these two partitioning steps. The binding component selects the task/resource combination to a given task first, and the scheduling component evaluates the scenario and acts as allocation index. With the help of task priorities, the most appropriate solution is chosen and the sequence of the task is set.

In this process, both the selection of the resource and the sequencing of task scheduling are performed. The following sections analyze the issues of *ReCA* with regard to the resource selection and sequencing of task scheduling.

Resource Selection

Constructive algorithms bind tasks to resources in respect of updated tasks priorities. For *ReCA*, the tasks priority updates are based on the prevailing schedule. Although the usage of the current schedule may improve the quality of the results, some situation still require further considerations.

A target architecture with two computational resources, such as a processor and a hardware accelerator, is given. An example of a simple CPG with annotated WCETs is depicted on the left hand side of the figure. Task 0 and task 2 can be only processed on the processor, whereas task 1 can utilize both resources. On the right hand side of the figure, two schedules are shown. The data transfer latencies are annotated at the edges of the task graph.

The upper half of figure 4.1 is the result of an algorithm considering only events in the current schedule. Task 1 has two alternatives: Either processed on the processor with a duration of 5 time units (T), or processed on the HW accelerator with a duration of 2T plus a transfer latency of 2T. Taking the solution with the highest priority, the HW accelerator is chosen to process task 1. To process task 2, a further data transfer of 3T is necessary.

The lower scheduling in figure 4.1 shows the result of an algorithm considering potential future events in addition. With the recognition of the necessity of the subsequent transfer after task 1, the processor is selected instead of the obvious faster HW accelerator. By avoiding the communication on the shared bus, the schedule results in a shorter overall latency. The consideration of future events in constructive algorithms can improve the performance and quality of the results.

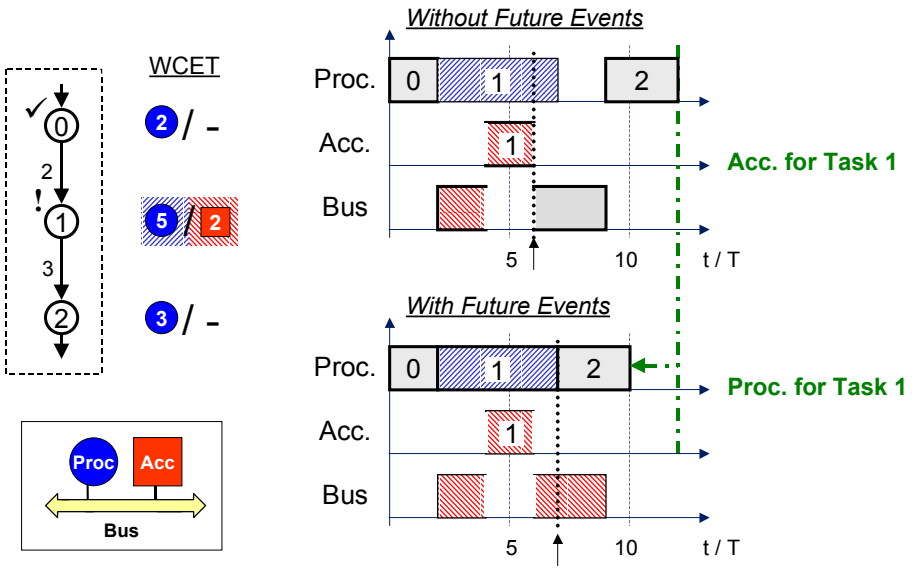


Figure 4.1: Communication Issue

Sequencing of Task Scheduling

In the preceding section, the constructive partitioning algorithm compares bindings of single tasks to different resource to find the most appropriate partition. Even if the bindings are selected in the first place, the sequence of task scheduling steps are important.

In an example shown in figure 4.2, the target architecture consists of two processors and the binding of the tasks is already selected. The communication is not considered in this example since it does not influence the effect to be shown. The only difference would be additional latencies by data transfers in the scheduling. In the case task 1 and 2 have the same task priority, list schedule algorithms do not differentiate between the handling of task 1 and 2. In addition, the data dependencies of node 2 and 3 usually remains undetected. By inspecting the CPG, it can be easily seen that processor 2 cannot start executing any tasks unless task 2 is finished. The right-hand side of figure 4.2 shows the two different scheduling possibilities of this given CPG. With task 1 first, the nodes 0, 1, and 2 are processed in sequence, before processor 2 performs the remaining tasks. With task 2 first, some parallel processing can be achieved. The earlier processing of task 4 reduces the total duration.

Although all tasks are already bound to resources and the binding is completed, the sequence of tasks selection is significant. This example should indicate that more information is needed for each scheduling step to determine the most appropriate solution and exploit the potential of the resources.

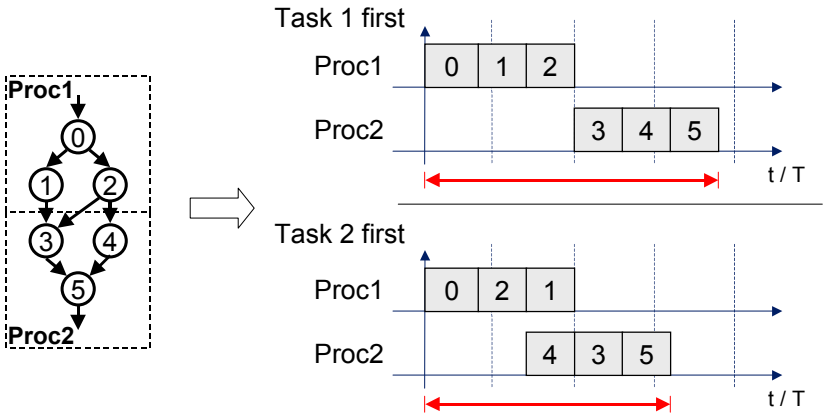


Figure 4.2: Importance of the Sequence of Task Scheduling

Clustering of Common Implementation Nodes

Some application specific features of design models require certain implementation constraints. Partitioning algorithms usually consider nodes as self-contained behavioral objects which may be individually bound to any available resource. If the design model is specified on a more fine-grain level of detail, special considerations need to be taken to avoid undesired conditions:

For instance, memory accesses are initiated by a certain resource which expect the data to be received. In the design model, the tasks before and after such memory accesses require to be bound to the same resource. Instead of having only one task to be processed during partitioning, a group of two tasks now represents the self-contained function. Functionalities composed of sub-tasks belonging together need to be understood as contiguous nodes with common binding characteristics. These nodes need to be collectively mapped to the same resource subset of target architecture. Such nodes are called Common Implementation Nodes (CIN).

The middle of figure 4.3 shows an example of a memory access. The CINs of task 1 and 3 should be bound to either a processor or a hardware accelerator, while task 2 is bound to memory representing a memory access. The four binding possibilities are placed around the task graphs. On the left hand side, solution I.) and II.) are allowed combination, since task 1 and 3 are bound to the same resource. However, solution III.) and IV.) on the right hand side mix up this requirement and create unacceptable solutions.

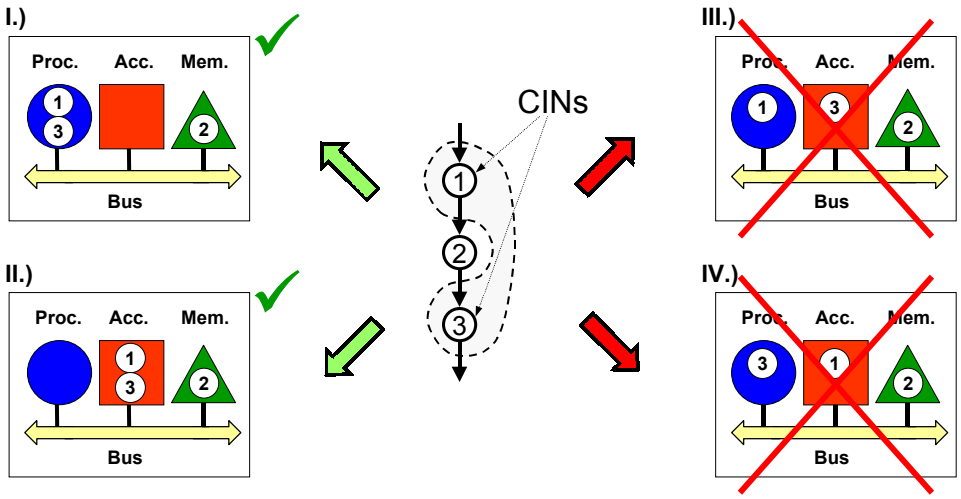


Figure 4.3: Binding Constraints for the Modeling of Memory Accesses with the Help of Common Implementation Nodes (CIN)

4.2 Look-Ahead

This section considers further details of the two algorithm improvements, resource selection and sequencing of tasks. The node priorities generated by list scheduling algorithms, such as *ReCA*, are assumed to reflect the actual importance of the single tasks regarding performance. However, these priorities are determined without an actual scheduling, or with the mere knowledge of previously scheduled events.

During partitioning, the task priority calculation significantly depends upon the prevailing schedule which has great influence on the binding selections. Moreover, slightly different schedules can cause completely different results. Therefore, the appropriate tasks need to be selected in a favorable sequence. The following algorithm extensions increase the amount of information used for task priority calculation. Potential future events may influence the partitioning decisions and increase the performance.

4.2.1 Resource Selection

The selection of the most appropriate resource is important to achieve satisfying results. The kind of information used for priority calculation may influence the outcome of the partitioning algorithm. In section 4.1, the influence of future events on performance is shown which may improve the results of constructive algorithms. In the following, the applicability of future events to improve the performance is evaluated.

Figure 4.4 shows a situation with three tasks and two computational resources. Resource 1 may be compared to a microprocessor while resource 2 with a better performance may represent a HW accelerator. The middle task is about to be partitioned to either resource. The execution latency of resource 1 is quite large compared to the latency of resource 2. Obviously, the faster resource 2 is selected. This decision is made without considering future events. By also considering mandatory future transfer latencies (here $t_{Transfer,2}$), the usage of resource 1, which is less performing and has a longer latency, results in a better overall performance.

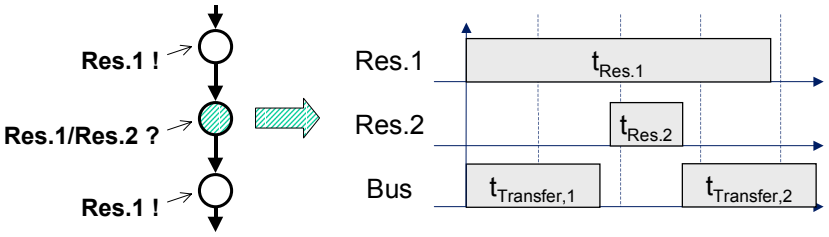


Figure 4.4: Consideration of Inevitable Communication for Partitioning

In the following, the priority calculation with and without regard to future events are evaluated. List scheduling algorithms, such as applied in *ReCA*, use the following condition for the selection of the faster resource of the target architecture, such as resource 2:

$$t_{Transfer,1} + t_{Res2} < t_{Res1} \quad . \tag{4.1}$$

The following condition takes mandatory future data transfers into account. The resource resulting in a better performance is selected, which is resource 1 in the example of figure 4.4. In (4.2), the inevitable subsequent data transfer is also considered:

$$t_{Res1} < t_{Transfer,1} + t_{Res2} + t_{Transfer,2} \quad . \tag{4.2}$$

The combination of (4.1) and (4.2) results in

$$t_{Transfer,1} + t_{Res2} < t_{Res1} < t_{Transfer,1} + t_{Res2} + t_{Transfer,2} \tag{4.3}$$

and is transformed to

$$0 < t_{Res1} - (t_{Transfer,1} + t_{Res2}) < t_{Transfer,2} \, . \tag{4.4}$$

If ($t_{Transfer,1} + t_{Res2}$) is greater than t_{Res1}, the algorithm without future events also selects resource 1 leading to no extra communication. If t_{Res1} - ($t_{Transfer,1} + t_{Res2}$) is greater than $t_{Transfer,2}$, the algorithm with future events is selecting resource 2. So does the algorithm without future events. In both cases, the algorithm without future events selects the same resource.

Provided that $t_{Transfer,1}$ and $t_{Transfer,2}$ are equal, (4.4) reduces to:

$$t_{Res1} - t_{Res2} < 2t_{Transfer} \tag{4.5}$$

Assuming a high t_{Res1}/t_{Res2}, the HW latency t_{Res2} can be neglected. Whenever the CPU latency is less than double the communication latency, the algorithm with future events is expected to perform well.

Future Events

The assumptions of potential future events influence the selection of task/resource combinations. However, these selections affect the actual taking place of the assumed future events. The prevailing schedule may prevent future events from being selected as expected. The further the events occur in the future, the less certain their taking place exactly as predicted.

In the following, only the immediate successors of a given task are considered which represents immediate data transfers and tasks, see figure 4.5.

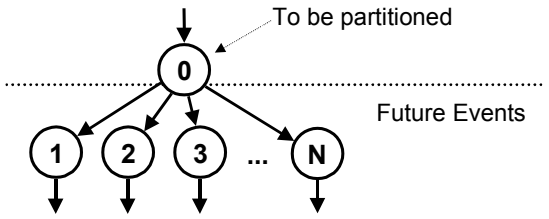


Figure 4.5: Consideration of Succeeding Data Transfers and Tasks as Future Events

N is the number of immediate succeeding tasks and R_i the number of possible resources for the i^{th} immediate succeeding node. The total number of different combinations C to execute the succeeding tasks is growing exponentially with the number of tasks

$$C = \prod_{i=1}^{N} R_i \quad \text{or } C = R^N \text{ with } R = R_1 = R_2 = \ldots = R_N. \tag{4.6}$$

Even for small numbers of tasks and resources, the probability to accurately predict the events in the future is low. However if the succeeding tasks have to be bound to different resources than the task to be partitioned, the corresponding data transfers can be predicted and used to improve priority calculation. Hence, the algorithms for resource selection introduced in the following solely consider the immediate succeeding transfers and the succeeding tasks.

4.2.2 Sequencing of Task Scheduling

The left task graph of figure 4.6 shows annotated global priorities which may be calculated by the HEFT algorithm, [93]. The priorities represent the latency of the critical path through the remainder of the task graph. This information is applied for selection of the most appropriate task/resource combination. Although task 1 and 2 have the same priority, the order of scheduling is significant to the result of the partitioning algorithm, as shown in figure 4.2.

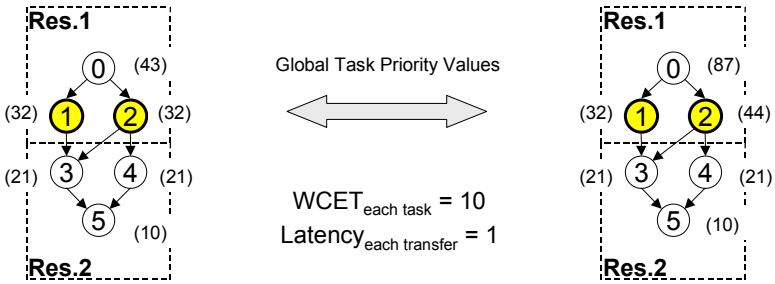


Figure 4.6: Issues of Priority Determination

In order to avoid this ambiguity, the global priority is determined by taking all succeeding branches into account. The effectiveness can be seen with task 2 which has a greater global priority in the right task graph of figure 4.6. In this way, all succeeding branches are regarded, and its priority is distinct.

In the next sections, the algorithm improvements for resource selection and sequencing of task scheduling are presented.

4.2.3 Algorithm Improvements for Resource Selection

The utilization of additional information during partitioning may benefit the performance of constructive algorithms. The consideration of potential future events and the following modifications of the partitioning algorithm are introduced by the Enhanced Constructive Algorithm (*ECA*).

Depending on each binding, constructive algorithms arrange for data transfers on demand. The communication resource connecting all computational resources is usually susceptible to become the bottleneck of the design due to lacking the planning of the communication. Since only computational resources and necessary data transfers are taken into account for priority calculation, the inevitable subsequent data transfers are neglected. This may result in degraded performance, while restricted binding possibilities give valuable information about potential communication bottlenecks.

To avoid such communication bottlenecks, *ECA* considers the additional offset `look_ahead`. This offset may include all immediate future events, such as the immediate succeeding transfer and the succeeding tasks. The implementation of *ECA* is based on *ReCA* with the additional *DU* value update which takes `look_ahead` into account for the priority calculation, see figure 4.7.

```
1.    calculate Static_Urgency for each node;
2.    while(tasks are left to be mapped)
3.    do
4.        determine all ready tasks;
5.        for (T = all ready_tasks)
6.            for (R = all possible resources)
7.                Dynamic_Urgency(T; R) = Static_Urgency(T) -
8.                    max(ready_data for T; R available) - WCET(T, R);
9.                Dynamic_Urgency(T; R) -= look_ahead();
10.           end
11.       end
12.       (Tmax,Rmax) = determine max Dynamic Urgency;
13.       bind_and_schedule(Tmax,Rmax);
14.   done
```

Figure 4.7: Extension of *ReCA* for *ECA*

For the analysis of Look-Ahead (*LA*), three different versions of the *ECA* considering future events are evaluated. One variant takes only inevitable data transfers into account, while the other two variants regard the immediate succeeding tasks and the necessary data transfers developing from the requirement of context transfers.

- *ECA_LA1* considers all immediate mandatory data transfers to the succeeding tasks. Whenever the task to be partitioned considers a resource which is not available for the immediate succeeding tasks, the corresponding context transfers are inevitable and the latencies of the data transfers are taken into account in the calculation of *DU*. In case the same resource is available for both the task to be partitioned and each succeeding task, the latency of the context transfer for each succeeding task will be ignored for priority calculation.
- *ECA_LA2* determines the offset `look_ahead` based on the shortest latency of the immediate succeeding data transfers and tasks. In this way, the fast succeeding resource should stimulate the selection of the same fast resource for the tasks to be partitioned. If the same resource is actually selected, the succeeding context transfer is not necessary. Otherwise, the data transfer is regarded in the `look_ahead` value. In addition, the execution latencies of the tasks are also accounted.

- *ECA_LA3* analyzes the subsequent tasks in a similar manner as *ECA_LA2*. Instead of determining the shortest latencies, the longest latencies of the immediate succeeding data transfers and tasks are taken into account. In this way, these succeeding events represent the worst case. To compensate for this worst case, the fastest resource is chosen for the task to be partitioned.

Figure 4.8 compares the three versions of *LA* on the basis of a simple example. In the upper part of the figure, a sample sub-CPG is given. The target architecture consists of two resources. Resource 1 is a slow performing resource, while resource 2 can execute the tasks faster. The sub-CPG on upper right-hand side depicts the implementation possibilities. Task 0 needs to utilize resource 1, while task 1 can only be processed by resource 2. Task 2 can choose between both resources. In the lower part of the figure, the three versions of *LA* are exemplified. The bold marked edges and nodes are taken into account for the `look_ahead` value.

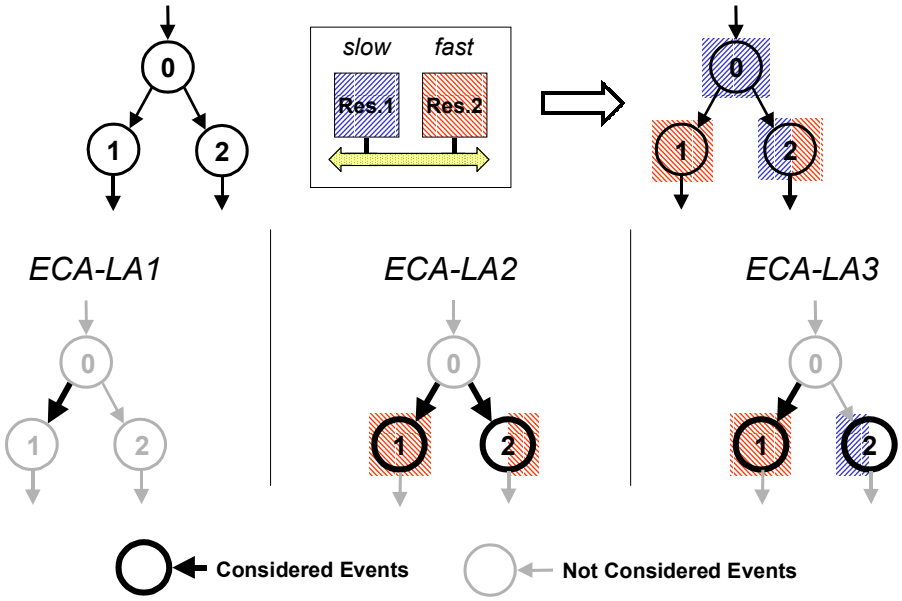


Figure 4.8: Variants of *ECA*

ECA_LA1: All Mandatory Transfers

This version of *ECA* identifies all subsequent mandatory data transfers. Mandatory data transfers caused by bindings to different resources are the only information which is assured depending on the selected resources. In target architectures with HW accelerators, such data and context transfers can occur quite frequently, since these HW accelerators can only process a fraction of the available tasks. Moreover at first sight, a shorter latency of the fast resource may look more favorable. However at a second glance, using a more

favorable resource with a longer execution time can avoid bus congestion and can result in a shorter overall latency as shown figure 4.1 on page 63.

The following pseudo code gives an outline of the functionality of the *ECA_LA1*.

```
1.    function look_ahead() // ECA_LA1
2.    {
3.        set look_ahead = 0
4.        for all succeeding tasks of node(node_sux)
5.            wcet_sux = WCET(node_sux, considered_resource)
6.            if ( wcet_sux is not available )
7.                set look_ahead += comm_latency
8.        end
9.        return(look_ahead)
10.   }
```

Figure 4.9: Outline of *ECA_LA1*

After the correction factor `look_ahead` is initialized, all succeeding nodes are examined and checked which resources are available. Whenever a succeeding task cannot be executed on the same resource as the current task, a context transfer and data transfer has to be performed. All mandatory transfer durations are summed up in `look_ahead`.

ECA_LA2: Succeeding Transfers and Tasks with the Shortest Latency

ECA_LA2 considers not only data transfers, but also latencies caused by succeeding tasks. In comparison to the mandatory data transfers of *ECA_LA1*, the binding of the succeeding task is not assured. Therefore, *ECA_LA2* assumes that the best performing combination of succeeding data transfer and task is chosen.

After the correction factor `look_ahead` is initialized, all succeeding nodes are examined. All possible resources for these nodes are compared to the current node. Whenever a succeeding task cannot be executed on the same resource as the current task, a data transfer needs to be performed. The total duration for this potential resource of a succeeding tasks is determined by the transfer duration and the WCET. Out of all total durations for this particular node, the shortest one is obtained and transferred to `look_ahead` .

ECA_LA3: Succeeding Transfers and Task with the Longest Latency

ECA_LA3 is similar to *ECA_LA2* except using the longest latency of the available resources. The motivation is to select the fastest possible resource now to compensate for less performing resources in the near future.

The pseudo code differs from the one of *ECA_LA2*, figure 4.10, in line 12 with the expression: `"set lat = determine max(latency_sux)"`.

```
1.    function look_ahead() // ECA_LA2
2.    {
3.        set look_ahead = 0
4.        for all succeeding tasks of node(node_sux)
5.            for all available resources(res)
6.                if (res == considered_resource)
7.                    set comm_latency = 0
8.                else
9.                    determine comm_latency
10.               wcet_sux = WCET(node_sux, res)
11.               set latency_sux = comm_latency + wcet_sux
12.               set lat = determine min(latency_sux)
13.           end
14.           set look_ahead += lat
15.       end
16.       return(look_ahead)
17.   }
```

Figure 4.10: Outline of *ECA_LA2*

4.2.4 Algorithm Improvements for Sequencing of Task Scheduling

The global task priorities or Static Urgencies (*SU*) used by *ReCA* only consider the latency of the critical path of subsequent events. The number of tasks which are about to be performed by each task is not used. In the following, an extension to *ECA* is introduced to take all succeeding processing paths of the CPG into account by modifying the *SU* calculation.

Enhanced Static Urgency Calculation

The Enhanced Static Urgency (*ESU*) represents an indication of processing capabilities necessary to perform the subsequent tasks. The SU value enables the comparison of two nodes regarding their critical path to the final node. This gives already an idea of their urgency to be scheduled as next task. However, branching is not considered in this priority by *SU*. Since the worst case causes all tasks to be processed by one single resource, the aggregation of all succeeding latencies represents a useful index.

Regarding the performance requirements, the difference to the *SU* calculation used by *ReCA* is using the sum of all succeeding paths, instead of comparing all paths and using the maximum latency. Figure 4.11 gives the outline of the *ESU* calculation.

```
1.    function ESU()
2.    {
3.        while (not all nodes are processed)
4.        do
5.            for all nodes(node)
6.                if (node == last node)
7.                    SU(node) = average_WCET(node)
8.                if (all successors are processed)
9.                    for all successors(sux)
10.                       SU'(sux) = SU(sux) + comm_latency
11.                       set SU(node) += SU'(sux)
12.                   end
13.                   set SU(node) += average_WCET(node)
14.                   mark node as processed
15.           end
16.       done
17.   }
```

Figure 4.11: Outline of the *ESU* Calculation

4.3 Clustering

Gajski et al. introduce specification requirements for embedded systems in [35]. The hierarchy of behavior objects is one of these requirements to ease the conception phase by hierarchically decomposing functionalities into a set of sequential and concurrent behaviors. However instead of breaking down functionalities into tasks, a way of pooling tasks with the same implementational constraints may be desired.

In this thesis, clustering is used as a possibility to enforce certain implementation characteristics for a group or cluster of tasks; hence the name clustering. Dave et al. defines the expression "cluster" in [22]: "A cluster of tasks is a group of tasks which are always allocated to the same PE (processing element)." These clusters reduce communication and therewith significantly speed-up the architecture synthesis process.

A different use of such clusters may help designers to allow design models with various levels of detail. The support of different levels of detail within one design model allows more flexibility during model creation. However, no information about potential clusters of tasks is available in neither the task graph nor the WCET table.

Partitioning algorithms usually expect nodes of task graphs as self-contained behavioral objects. Such self-contained tasks are individually handled and bound to resources. In situations, such as data interaction to and from the memory which should be initiated and handled by the same computational resource, undesired resource assignments may occur.

Such intermediate memory accesses between behavioral objects may be bound to different resources by the partitioning algorithm.

Nodes with special implementation constraints can avoid undesired resource combinations. Such nodes are called Common Implementation Nodes (CIN) in the following. Grouped in a cluster, CIN are bound as a whole to any available resource. Clusters and CIN may be applied for implementation trade-offs performed during partitioning on the fly.

4.3.1 Different Design Objectives

Different implementation approaches may solve performance issues. Improvement in performance can be achieved by accelerating tasks with HW accelerators, or by exploiting parallelism distributing functions on multiprocessor systems. The exploration of various implementation possibilities within one design model may be supported by design objectives. Design objectives can place special emphasis on the outcome of the analysis, such as parallel processing or the acceleration of certain functionalities. By clustering CIN, design models may be annotated to accommodate different design objectives.

For these different design objectives, each implementation approach demands to map its characteristic to an own analysis model resulting in different analysis models. Generating analysis models for each different way of implementation is a very complex and time consuming task. Hence, merging of these different design approaches in one single analysis model would be preferable.

Resource sets help to compare various target architectures using the same analysis model. Model annotations allow the partitioning algorithms to differentiate between the various implementations. Nevertheless, minor modifications of the CPG are necessary to align the common model for the different implementations, since each implementation possesses a distinct internal structure processing the data in a different way.

Figure 4.12 shows the usage of a single analysis model with two different design objectives. In the middle of the figure, a CPG illustrates the functionalities of a given application. The left and right representation of the figure depicts the same CPG with clusters of CIN which are bound to resources. With the design objective of concurrent execution shown on the left representation, all CIN of the depicted clusters are bound to corresponding resources and the corresponding schedule can be determined. For the right representation, the design objective is applied to speed up the total performance by accelerating certain functionalities. The target architecture represents the design objective by having two different HW accelerators.

With the help of clustering, different design objectives can be applied and evaluated based on a common design model.

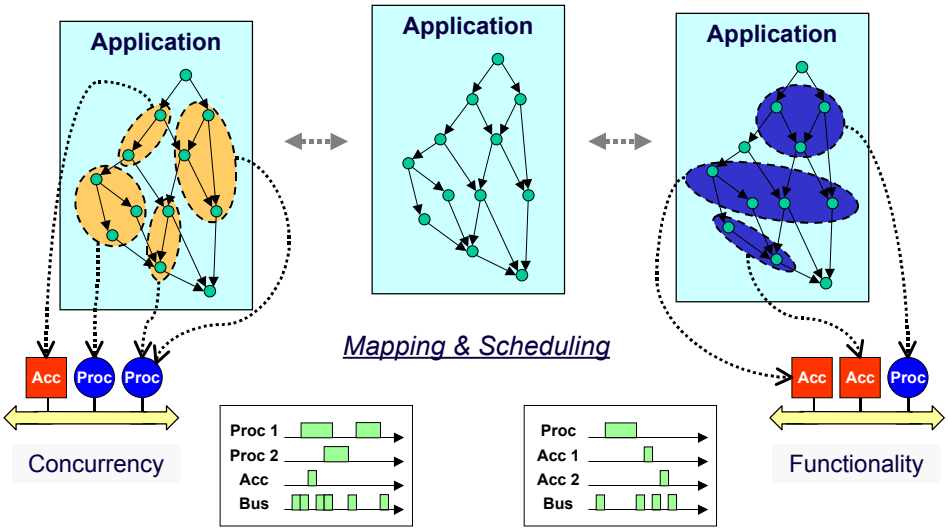


Figure 4.12: Different Design Objectives for the Exploitation of Concurrency or Acceleration of Functionality Supported by Clustering

4.3.2 Resource Sets for Common Implementation Nodes

The previous sections introduce clustering for the binding to one single resource. Extending the definition of cluster by Dave et al., the use of clusters is not only restricted to the same processing element, but expanded to sets of resources. Various implementation possibilities which are not limited to one single resource each are allowed. Such resources of a target architecture specify a resource set for each implementation possibility provided that the different level of granularity allows a common analysis model.

Resource sets may be utilized for the immediate comparison of implementation possibilities during partitioning. Other partitioning methodologies require several design models for the evaluation of each implementation possibility. An example of different implementation possibilities are demonstrated in figure 4.13.

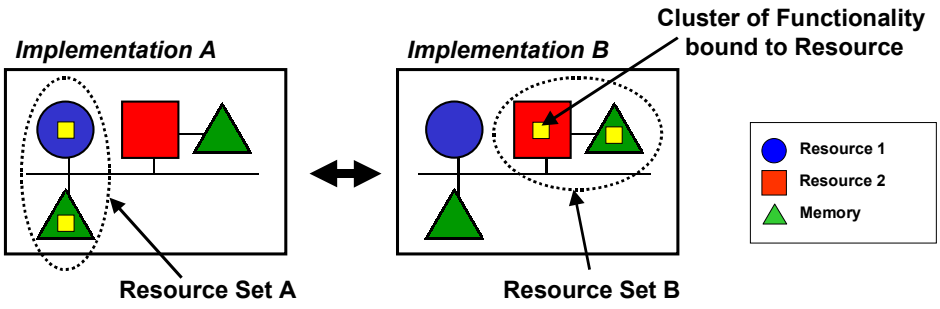


Figure 4.13: Different Implementation Possibilities

A given functionality, depicted as small squares, can be implemented with two different resource sets. Implementation *A* is represented by resource 1, which may be a

microprocessor, and shared memory. Resource 2 and a local memory form implementation *B* which may be a HW accelerator block. In this example, it can be seen that the cluster is not only bound to one single resource, but to a group of resources belonging together. In this way, the interaction between the tasks, for instance, bound to the processor and the access to other resources, especially the memory, may be analyzed.

In the following, the algorithms for the selection of the resources needs to be extended to cover clusters in the priority calculation.

4.3.3 Algorithm Improvements for Common Implementation Nodes

The partitioning of CIN belonging to a cluster gives information about the future bindings of all other CIN of this cluster. This information of future bindings are used for priority calculation. Hence, this section proposes an extension to *ECA* to compare different resource sets for each cluster by considering special binding constraints.

An approach to consider different resources mutual exclusively for CIN is introduced in [8]. The WCET table represents the target architecture with all available resources for each behavioral object. Since no information about the desired relationships between tasks is available, an additional CIN list per cluster is necessary to identify self-contained functionalities forming a cluster of CIN. The additional information about the clusters help to prevent undesired binding selections across the different implementations at the same time.

Figure 4.14 shows two different implementation alternatives with a resource set each. The given WCETs of the example represent the characteristics of the used resources. Resource set "SW" with shared memory, which may represent a microprocessor, and resource set "HW" using internal memory, may be an application-specific hardware accelerator. The cluster of CINs contains all tasks representing the self-contained functionality allowing these tasks to be bound to either resource set "SW" or resource set "HW". $Cluster_1 = \{0; 1; 2\}$ indicates that all tasks in this example are affected.

Whenever one Common Implementation Node of a cluster is selected, the other CIN are assigned to the same resource set. The selection of the resource set "HW" specifies that, once one task of the cluster of CINs is mapped to the resource 2, all other tasks of the cluster of CINs also have to be mapped to resource 2. Alternatively, if the resource set "SW" is selected, the tasks may be bound to whatever component of the resource set "SW" is available.

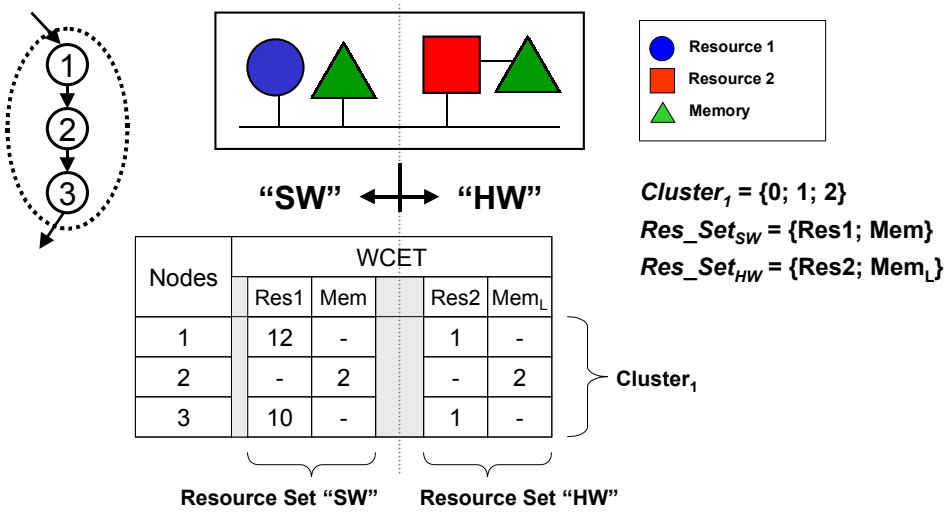


Figure 4.14: Clustering with Common Implementation Nodes (CINs)

In this way, particular resource constellations may be reproduced without restricting the selection to one resource. Of course, the WCET table may be extended to allow the availability of several components within one resource set for one behavioral object. This mechanism makes it possible to compare different ways of implementation within the same analysis model.

Algorithm Extensions of *ECA*

ECA is extended to properly process CIN of clusters with implementation constraints and to take future binding restrictions into account. Figure 4.15 outlines the modifications to *ECA* with bold expressions. If the considered cluster is already assigned, line 7 to 9 of the pseudo code ignores all resources not belonging to the corresponding resource set to enforce implementation constraints. The task priority calculation in *ECA* includes additional information about CIN within the cluster. Line 14 and 15 determine the parameter `look_ahead_cluster` which may include latencies of tasks and data transfers of the considered clusters. The *DU* value is updated with the resulting value of the parameter `look_ahead_cluster`.

The first CIN to be partitioned decides about the selected resource set. The WCET of the first task may delude about the performance of the entire cluster using the resource set. All other CINs of the cluster also need to be considered for the selection of the resource set to represent the performance of the resource set properly.

For flexible design modeling, the shape of clusters within a task graph does not need to be contiguous. Any CIN belonging to a specific cluster may be connected to any other tasks of the task graph. Therefore, any calculated LA values based on contiguous CIN, such as

```
1.    calculate Static_Urgency for each node;
2.    while(tasks are left to be mapped)
3.    do
4.        determine all ready tasks;
5.        for (T = all ready_tasks)
6.            for (R = all possible resources)
7.                if (T is CIN of a cluster &&
8.                        R is not contained in assigned resource_set)
9.                    skip R
10.               Dynamic_Urgency(T; R) = Static_Urgency(T) -
11.                   max(ready_data for T; R available) - WCET(T, R);
12.               calculate(look_ahead);
13.               Dynamic_Urgency(T; R) -= look_ahead;
14.               if (T is CIN of cluster)
15.                   Dynamic_Urgency(T; R) -= look_ahead_cluster();
16.           end
17.       end
18.       (Tmax,Rmax) = determine max Dynamic Urgency;
19.       bind_and_schedule(Tmax,Rmax);
20.   done
```

Figure 4.15: Clustering Extension for *ECA*

the critical path, may not properly represent the intended latency. Based on this consideration, two options are proposed in this following:

- The option *Cluster_CIN* depends on the first CIN of the cluster to decide about the used component of the utilized resource set. This option merely assures the implementation constraints. Since no further CIN is taken into account, the implementation is simple with the value 0 for the parameter `look_ahead_cluster`, see figure 4.16.

```
1.    function look_ahead_cluster() // Cluster_CIN
2.    {
3.        set look_ahead_cluster = 0
4.        return(look_ahead_cluster)
5.    }
```

Figure 4.16: Calculation of `look_ahead_cluster` for *Cluster_CIN*

- The option *Cluster_Sum* takes all CIN of a cluster into account for priority calculation. For all other CIN of the cluster, the available resource of the resource set is selected. The WCETs of the CIN are added and the result is assigned to the parameter `look_ahead_cluster`, see in figure 4.17.

```
1.    function look_ahead_cluster() // Cluster_Sum
1.    {
1.        set look_ahead_cluster = 0
2.        determine cluster_list[] with T as member
3.        set res_set = determine_resource_set(R)
4.        for (next_task = all CIN of cluster_list)
5.            determine shortest_WCET(next_task, res_set)
6.            set look_ahead_cluster += shortest_WCET
7.        end
8.        return(look_ahead_cluster)
9.    }
```

Figure 4.17: Calculation of `look_ahead_cluster` for *Cluster_Sum*

4.4 Performance Evaluation of Look-Ahead

In the following section, the performance capabilities of the variants of *ECA* are evaluated in relation to *ReCA*. Similar to chapter 3, classes of design models and architectures are represented by parameter sets. TGFF, [24], generates task graphs and target architectures based on these parameter sets. The used simulation environment is the same as in chapter 3. Each working point in the result figures represents the mean performance of the considered algorithm based on 100 synthetically generated task graphs and associated target architectures.

After the design model and architecture assumptions are specified, a performance evaluation of *ECA_LA1* regarding inevitable communication events is carried out. Then, the analysis of runtimes of the introduced algorithms, and the evaluation of the *ECA* variants used in multiprocessor systems are performed.

4.4.1 Design Model and Architecture Assumptions

To achieve the performance required for certain high-performance applications in networking, multiprocessor systems with dedicated hardware accelerators are increasingly used on a SoC. The evaluations in this thesis focus on such multiprocessor architectures.

The considered design models consist of 100 nodes in the average. Each node has 5 preceding nodes and 5 succeeding nodes in the mean. Regarding conditional branches, 10% of the nodes within the CPG are designated conditional. Out of these conditional nodes, 80% of the succeeding edges are assigned as conditional branches. The shared memory is accessed by 15% of the nodes according to CommBench, [104].

The target architecture contains two type of resources. Resource type A represents resources which can process all kinds of tasks, such as microprocessors. All instances of

resource type A are identical. Resource type B process only certain selected tasks of the application (here 25%) with a mean speed-up factor of 5 faster than resource type A. These selected tasks are arbitrarily chosen. Such resources may represent application-specific accelerators. Each instance of resource type B represents an individual resource which has its own characteristic. The components of the target architecture are connected by a common shared bus. The mean amount of data transferred over the common bus accounts for 64 bytes, if not overridden by the analysis boundary conditions.

The following table 4.1 represents a typical set parameter for potential scenarios used for the analysis later in section. The following constraints are applied to the tests as long as no other constraints are given:

Parameter	Value Range
Design Model	
Size of Graph	100
Number of Preceding Branches per Node	5
Number of Succeeding Branches per Node	5
Ratio of Conditional Nodes	10 %
Ratio of Conditional Edges originate from a Conditional Node	80 %
Ratio of Memory Nodes	15 %
Target Architecture	
Number of Resources on Resource Type A (homogenous)	2
Number of Resources on Resource Type B (heterogeneous)	4
Speedup of Resource Type B compared to Resource Type A	5:1
Ratio of executable Nodes on Resource Type A	100 %
Ratio of executable Nodes on Resource Type B	25 %
Number of Buses	1
Average Data over Bus	64 bytes

Table 4.1: Simulation Properties of CPGs and Target Architectures for the Performance Evaluation of *ECA*

4.4.2 Results

Performance of ECA_LA1 with regard to Inevitable Communication Events

From section 4.2.1, equation eq. (4.5) expresses the condition in which *ECA* is assumed to perform efficiently:

$$t_{CPU} - t_{HW} < 2 \cdot t_{Transfer} \qquad (4.7)$$

This equation expresses that the additional longer latency of the slower resource should be less than the double the data transfer provided that the data transfers to and from the resource take the same amount of time.

With a HW speed-up factor of 5, the mean latencies of the processors and accelerators relate to

$$t_{CPU} = 5 \cdot t_{HW}. \tag{4.8}$$

Eq. (4.8) inserted in eq. (4.7) results in the range where *ECA* is expected to perform well:

$$t_{CPU} < 2.5 \cdot t_{Transfer}. \tag{4.9}$$

This estimation is evaluated with *ECA_LA1*, since its LA calculation exclusively considers inevitable data transfers. In figure 4.18, the boundary of the made estimation is depicted by the bold rectangle (A). The x-axis represents the mean processor latency of all nodes in the CPG, while the y-axis stands for the mean transfers latency. The results are illustrated in the z-axis as mean expectations of the ratio of the latency determined by *ECA* to the latency determined by *ReCA*.

Two arrows show the region with increasing improvement capabilities. With increasing latency of the data transfer, *ECA* selects the more favorable resources resulting in a shorter overall-latency for the same design model. The variation of the WCETs and data transfers latency generated synthetically allow *ECA* to achieve better results. More heterogeneous target architectures induces more resource switching and may cause bottlenecks which may be resolved by *ECA*. In this scenario, *ECA* exceeds 35% reduction in mean overall latency compared to *ReCA*. This applies to the working point with a mean CPU latency of 50 time units and a mean transfer latency of 50 time units. These values easily meet equation eq. (4.9).

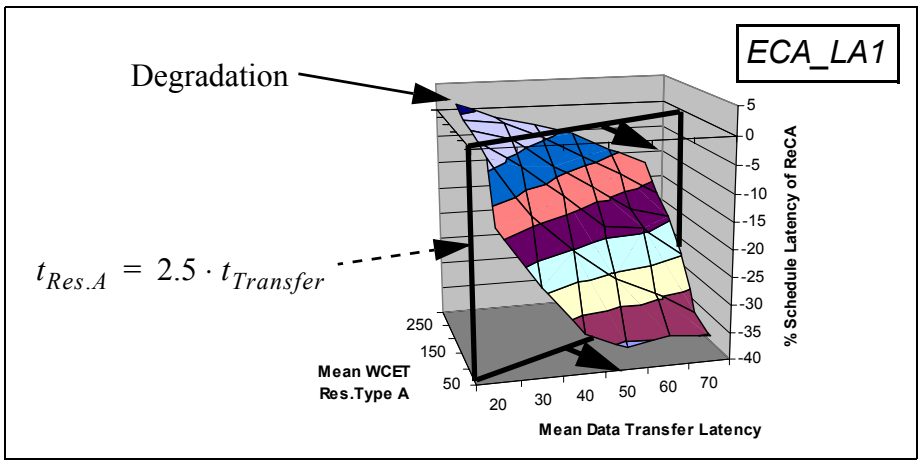


Figure 4.18: Performance of *ECA_LA1* Compared to *ReCA* Considering Inevitable Data Transfers

Since the task graphs are manifold in shape and structure, *ECA* cannot perform well for all scenarios and parameter sets. In the figure, there is an area where *ECA* produces worse results than *ReCA*. In these cases, *ECA* selects a task and a resource which is favorable in this situation, but it turns out that this selection is disadvantageous for the overall latency. The parameter sets for performance degradation are outside of the proposed design space by eq. (4.9).

Simulation Runtimes of ECA

The iterative heuristics shown in chapter 3 requires long simulation runtimes. Especially, the handle of larger task graphs is not practicable. Table 4.2 shows the runtimes for the introduced variants of the *ReCA* and *ECA*. The magnitude of the runtimes allow fast execution of partitioning even for very large CPGs. The variances of the shown results are caused by different sequences of comparing and scheduling the tasks.

ECA_LA1 performs with similar runtimes than *ReCA*. For each task to be scheduled, the corresponding resources of the succeeding tasks are checked. *ECA_LA2* and *ECA_LA3* requires to find the shortest respective longest latency of the succeeding task. All task/ resource combinations of the succeeding tasks need to be evaluated. Hence, the runtimes are increased in relation to *ECA_LA1*.

It can be observed that the calculation of *ESU* (Enhanced Static Urgency) consumes more time than the calculation of *SU*. The calculation of *SU* compares different succeeding path to determine the critical path, while *ESU* adds up all latencies. Nevertheless, the difference of both ways calculating the SU value scarcely influence the overall runtimes.

Size of CPG	**Runtime**							
	ReCA	***ECA_... without ESU***			***ECA_... with ESU***			
		LA1	***LA2***	***LA3***	***LA1***	***LA2***	***LA3***
100	0.04s	0.04s	0.04s	0.04s	0.04s	0.04s	0.05s
200	0.31s	0.31s	0.32s	0.33s	0.32s	0.33s	0.34s
300	1.20s	1.20s	1.26s	1.24s	1.37s	1.42s	1.40s
500	6.94s	7.31s	7.63s	7.71s	8.05s	8.11s	8.22s
1000	71.34s	73.06s	75.40s	75.34s	80.52s	81.42s	81.38s

Table 4.2: Simulation Runtimes of *ReCA* and the Variants of *ECA* in Comparison to the Size of the Design Model

Variants of ECA applied to Multiprocessor Systems

This section compares the results of the different variants of *ECA* based on the parameter sets of CPGs and target architectures, shown in section 4.4.1. Figure 4.19 depicts performance figures of this environment setup for the three versions of *ECA*, *ECA_LA1*, *ECA_LA2*, and *ECA_LA3*, in combination with and without *ESU*. Each chart represents the results in percentage of the schedule latency of *ReCA*.

ECA_LA1 provides significant reductions of the scheduling latencies up to 35%. This can be at the short WCETs of the less performing resource type A. With larger WCET of resource type A, the improvement capabilities are reducing, since *ReCA* also selects favorable resource more often.

By considering only inevitable data transfers, the eq. (4.5) can be written as

$$t_{Res.A} - t_{Res.B} < 2t_{Transfer} \tag{4.10}$$

and with

$$t_{Res.B} = \frac{t_{Res.A}}{speedup} \tag{4.11}$$

be transformed to

$$t_{Res.A} < 2t_{Transfer} \cdot \left(\frac{speedup}{speedup - 1}\right). \tag{4.12}$$

The higher the value *speedup*, the less important it becomes for influencing the improvements of *ECA_LA1*. With a constant mean value of data transfer latency, eq. (4.12) and the results show that the performance improvements of *ECA_LA1* are hardly influenced by the speed-up factor of resource type B. The influence of *ESU* is not significant on *ECA_LA1*. The consideration of known immediate succeeding data transfers is much more dominant than the resolution of priority ambiguity.

ECA_LA2 delivers degraded performance than *ECA_LA1*. The succeeding tasks are assumed for priority calculation. Since the presumed assignments of future tasks may change, the selected resource may become unfavorable in the overall schedule. With the uncertainty of the expected schedule, the performance is about 7% less effective than *ECA_LA1*. With *ESU*, the performance of *ECA_LA2* can be improved. Since *ESU* considers all succeeding paths, the importance of the single tasks is more accentuated. In this way, more urgent tasks are processed earlier. With all immediate succeeding tasks and data transfers of *ECA_LA2*, the *ESU* summing up all succeeding paths allows more favorable selections of resources which allows up to 2% better results than not using *ESU*.

ECA_LA3 pursue a similar intention like *ECA_LA2* by using the longest latency of the succeeding tasks and data transfers. In this way, a fast performing resource was intended to be selected by *ECA_LA3*. However, *ECA_LA3* tends to select resources which avoid

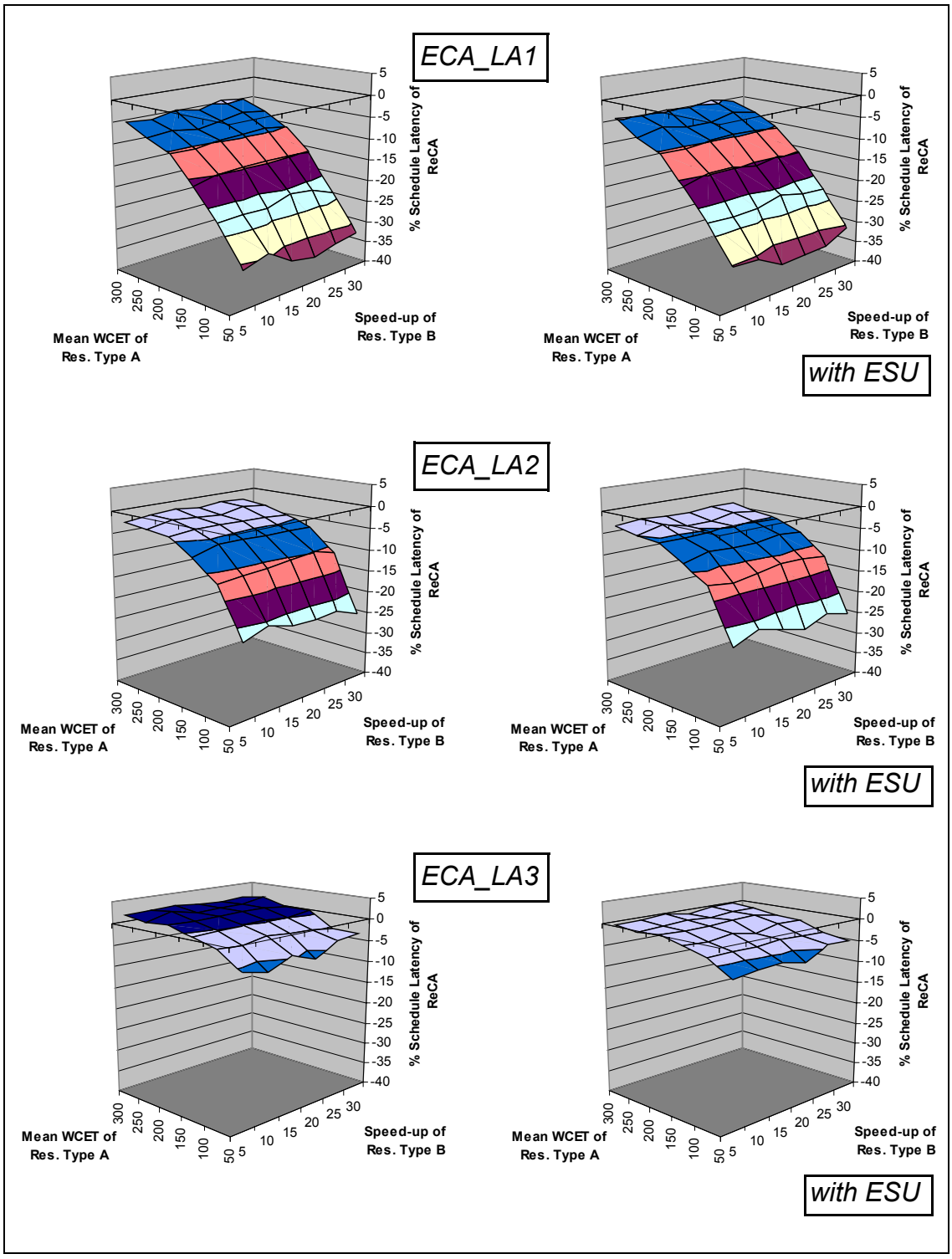


Figure 4.19: Performance Analysis for the Multiprocessor Scenario Relative to *ReCA* Based on the Parameter Set of Table 4.1

communication on the shared bus. By considering slow performing resource for future events, *ECA_LA3* also selects slow resources for the considered task. Hence, the overall performance of the algorithmic improvements are marginal. In this example similar to *ECA_LA2, ESU* can improve the mean of schedule latency reduction by up to 3.5%.

The algorithmic modifications of *ECA* may also cause confusion in the scheduling sequence and resource assignment and may result in worse performance than *ReCA*. For *ECA_LA3* without *ESU*, such an area of degradation starts early in comparison to the other algorithms with a worst performance of a schedule latency extension of +1.8% compare to *ReCA*. *ECA_LA3*'s low performance can be mitigated by *ESU* to avoid degradation by the enhancement. Nevertheless due to the weak performance of *ECA_LA3*, only *ECA_LA1* and *ECA_LA2* are allowed for further investigations.

4.5 Performance Evaluation of Clustering

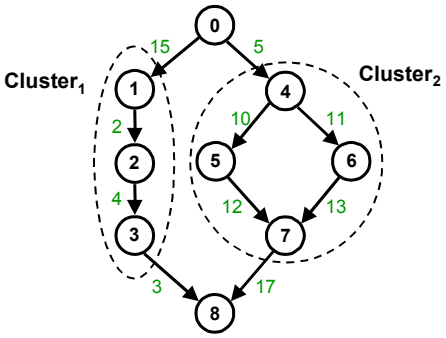
For the evaluation of clustering, the simulation environment needs a modification. During the creation of task graphs, TGFF creates CPGs with arbitrary connections among the nodes. To facilitate the generation of self-contained groups of tasks, clusters of CINs may be inserted in the main task graph in the form of small sub-CPGs. The self-contained functionalities are represented by a cluster of CIN with one start node within a cluster receiving all necessary input data and one end node passing on the processed data to the succeeding tasks. In this way, the integration of self-contained functionalities is facilitated and the way of refining the design model by a designer may be represented.

After the introduction of the modified analysis environment, the utilized parameter set is presented. The performance capabilities of the clustering variants are analyzed showing the advantages of using the *Cluster_Sum*. Then, the *LA* variants of *ECA* are evaluated in combination with the two clustering algorithmic modifications.

4.5.1 Modification of Evaluation Environment

The generation of task graphs with self-contained functionalities represented as clusters of tasks requires a multi-stage procedure. Instead of generating the task graph at once, smaller self-contained functionalities task graphs are created and inserted in the main task graph, [60]. In this way, the refinement of single functionalities with more details can be represented.

Figure 4.20 shows an example of a task graph including two clusters of tasks which are executable on corresponding accelerators. Here, task 1, 2, and 3 incorporate cluster 1. These tasks can be processed either on the processor or the accelerator 1. For cluster 1, the input data are provided from tasks 0 by a data transfer, if a context transfer is necessary, and the output data are provided to task 8. No data transfers to and from the inside of the



WCET Table

Node	Proc 1	Proc 2	Acc 1	Acc 2	
0	**2**	**10**	-	-	
1	**20**	**80**	**6**	-	Cluster$_1$
2	**30**	**90**	**10**	-	
3	**40**	**100**	**13**	-	
4	**50**	**110**	-	**12**	Cluster$_2$
5	**60**	**120**	-	**15**	
6	**70**	**130**	-	**17**	
7	**80**	**140**	-	**20**	
8	**5**	**20**	-	-	

Figure 4.20: Required Outcome

cluster are supported to facilitate the generation and the analysis in this thesis. However, the algorithm may process arbitrary connected CINs.

For this example, the analysis environment generates three task graphs: a main task graph including task 0 and 8, and two task graphs representing cluster 1 and 2. These three graphs are merged and applied to *ECA*.

4.5.2 Design Model and Architecture Assumptions

Similar to the analysis of *ECA* in section 4.4.1, the parameter are set for the analyses of the performance of *ECA*. The size of the task graphs is doubled to accommodate four clusters of tasks each with a size of 30 nodes. The target architecture again consists of homogeneous resources type A, such as microprocessors, and four resources type B, for instance dedicated hardware accelerators, with a speed-up of 5:1. The ratio of conditional nodes is 10%.

Properties	Value Range
Design Model	
Size of Total Graph	200
Number of Clusters	4
Size of each Cluster	30
Number of Preceding Branches per Node	5
Number of Succeeding Branches per Node	5
Ratio of Conditional Nodes	10 %
Ratio of Conditional Edges originate from a Conditional Node	80 %
Ratio of Memory Nodes	15 %

Table 4.3: Simulation Properties of Graph and Target Architecture for the Performance Evaluation of *ECA* Extensions for Clustering

Properties	Value Range
Target Architecture	
Number of Resources on Resource Type A (homogenous)	2
Number of Resources on Resource Type B (heterogeneous)	4
Speedup of Resource Type B compared to Resource Type A	5:1
Ratio of executable Nodes on Resource Type A	100 %
Ratio of executable Nodes on Resource Type B	25 %
Number of Buses	1
Average Data over Bus	64 bytes

Table 4.3: Simulation Properties of Graph and Target Architecture for the Performance Evaluation of *ECA* Extensions for Clustering

4.5.3 Results

For the evaluation of the performance capabilities of clustering for introduced constructive partitioning heuristic, *ReCA* is extended by the clustering functionality with the two options *Cluster_CIN* and *Cluster_Sum*. In this way, the effects of clustering can be analyzed without blending the effects of taking immediate succeeding events.

Consideration of Clustering

The cluster enhancements for *ECA*, *Cluster_CIN* and *Cluster_Sum*, are compared in figure 4.21. The x-axis represents the average WCETs of the resource type A while the speed-up factors of the resource type B are plotted on the y-axis. The results are depicted along the z-axis as ratio of the latency of *ReCA* with *Cluster_Sum* in relation to *ReCA* with *Cluster_CIN*.

The curve shows that *Cluster_Sum* produces results with up to 15% of schedule latency reduction in this scenario especially for short WCETs of resources type A. However for larger WCETs of resource type A, the performance improvements of *Cluster_Sum* are not that distinct.

The resource selected for the first CIN of a cluster is mandatory for the entire cluster. *Cluster_CIN* relies on the first CIN to determine the appropriate resource and does not take other CINs of the cluster into account. If this choice is unfavorable, other following CIN need to use this same resource. In this way, the overall performance may be impaired.

By reducing the priority of the CINs of a cluster, the process of partitioning the entire cluster is delayed. *Cluster_Sum* is taking all CINs of the cluster into account for the task priority update. Other tasks that are not contained in the cluster are processed earlier by the partitioning algorithm and can use the resource first which would have been allocated otherwise by the cluster. Later in the scheduling process, the CIN are bound to another resource being more favorable for the overall performance.

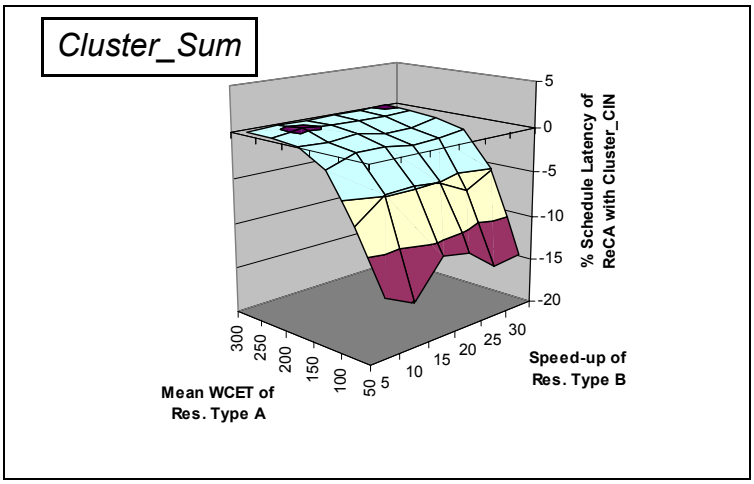


Figure 4.21: *ReCA* with *Cluster_Sum* Compared to *Cluster_CIN*

Similar to *ECA*, the results are not significantly affected by the speed-up factor of the resource type B. *ReCA* with *Cluster_CIN* always selects the resource with shorter WCETs. *ReCA* with *Cluster_Sum* behaves in the same way. Hence, no significant difference in the results can be demonstrated.

Simulation Runtimes of Clustering

The runtimes for the options of clustering are similar to the runtimes of *ECA*, see table 4.4. However, the following factors influence the runtime of the algorithms: the limited number of available resources for CINs, the calculation of the parameter `look_ahead_cluster`, and the number of tasks ready to be scheduled.

Although the parameter `look_ahead_cluster` needs to be determined anytime a CIN is ready to be scheduled, the number of such CINs is small for CPGs with a low quantity of nodes. The number of cycles in which certain tasks are compared with other tasks is small. Moreover, the algorithms with cluster options show shorter runtimes for small CPGs than *ECA*, because the preselection of a resource within a cluster renders the evaluation of the other resources unnecessary.

For larger CPGs, the main reason for longer runtimes is the larger number of tasks to be scheduled. Each time a CIN is compared, the parameter `look_ahead_cluster` needs to be determined over and over again until the CIN is partitioned. Here, the reduced number of resources per CIN cannot compensate the longer runtimes.

Size of CPG	Runtime	
	Cluster_CIN	Cluster_Sum
100	0.02s	0.03s
200	0.12s	0.19s
300	0.58s	0.66s
500	7.34s	7.40s
1000	109.84s	109.94s

Table 4.4: Simulation Runtimes of *ReCA* with the Options for Clustering *Cluster_CIN* and *Cluster_Sum* in Comparison to the Size of the Design Model

Applying LookAhead on Cluster_CIN

Figure 4.22 shows the results of using *ECA* with *Cluster_CIN* applying *ECA_LA1* and *ECA_LA2* in relation to *ReCA* with *Cluster_CIN*. The charts show that neither variants of *ECA* can achieve improvements compared to *ReCA* with *Cluster_CIN*.

ECA_LA1 avoids communication on the shared bus by selecting the same resource of the preceding node, especially for short WCETs of resource type A. Since the fast performing resources of type B are only available for clusters, only resources of type A are chosen for execution. Hence, the overall latency is impaired by slow performing resources compared to *ReCA*. *ECA_LA2* exhibits similar performance figures.

ESU influences the outcome in the same way than not using clustering. The results of *ECA_LA2* can mitigated by improving the schedule of tasks not belonging to a cluster by up to about 2% of schedule latency. Also, *ECA_LA1* and *ESU* hardly show any differences in results.

Applying Look-Ahead on Cluster_Sum

In figure 4.23, the results of using *ECA* with *Cluster_Sum* are depicted applying LA with *ECA_LA1* and *ECA_LA2* compared to *ReCA* with *Cluster_CIN*. The charts show that with the usage of *Cluster_Sum* improvements of up to 15% can be achieved.

It can be observed that all variants of *ECA* with *Cluster_Sum* exhibit improved results up to about 3.5% for mid-range WCETs between 100T to 200T. For the given scenarios, *ECA_LA1* cannot achieve the same performance than in figure 4.19. The design model consists of two slow-performing homogeneous resources for all tasks and four fast-performing heterogeneous resources which are allotted to clusters. Between the two homogeneous resources, no inevitable data transfers takes place between consecutive computational tasks outside of clusters which *ECA_LA1* would take advantages of.

Some nodes of the design model represent memory accesses. These tasks can exclusively be assigned to a dedicated memory resource. If a future event is such a memory access, the succeeding data transfer will be inevitable. For such cases, *ECA_LA1* can

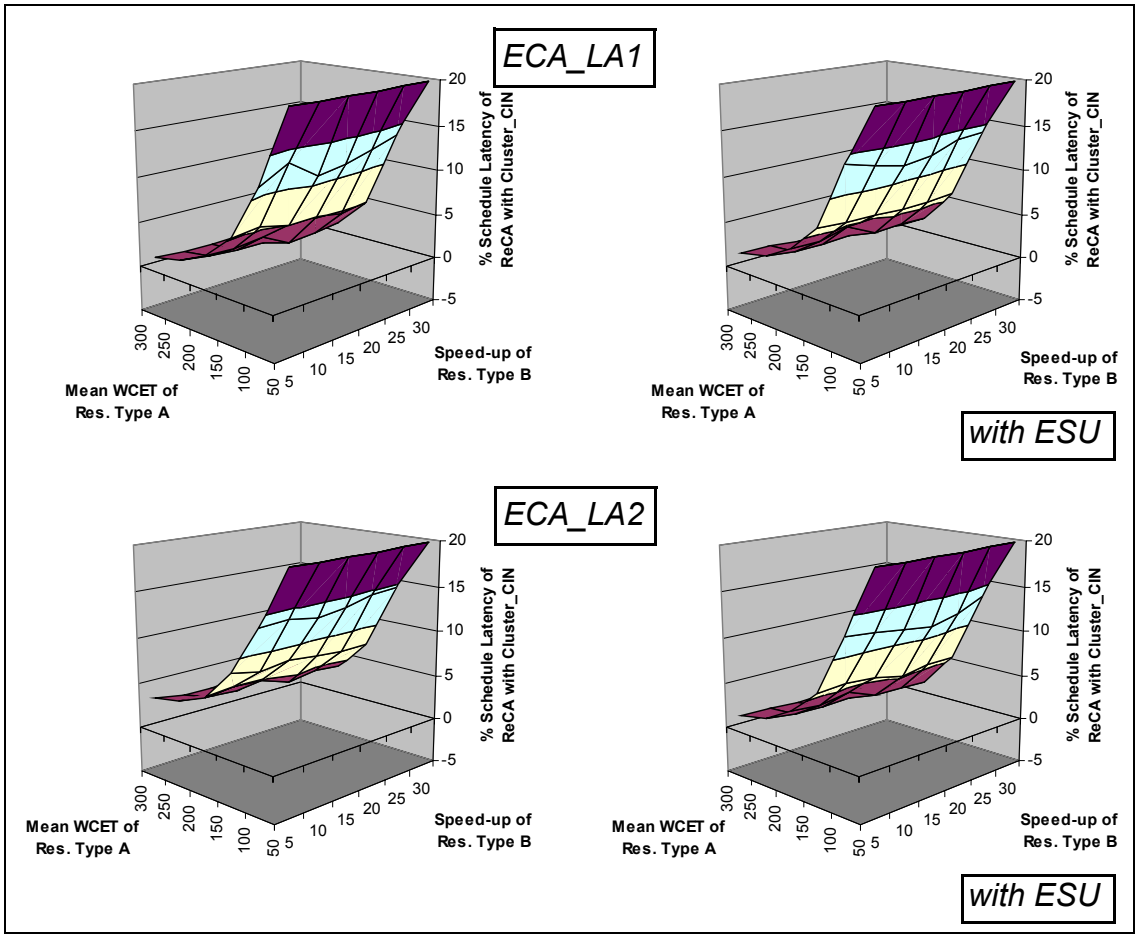


Figure 4.22: Performance Analysis of *ECA* with *Cluster_CIN* Relative to *ReCA* with *Cluster_CIN* in a Multiprocessor Scenario

improve the overall latency up to 2%. *ECA_LA2* can also process the design models efficiently. However, the consideration of speculative events cannot outperform *ECA_LA1*. Especially for large WCETs of resource type A, the algorithm cannot select the most appropriate resources and results in worse schedule latencies.

4.6 Summary, Comments and Conclusions

This chapter proposes enhancements for the Reference Constructive Algorithm (*ReCA*). Two main partitioning decisions can be identified as the resource selection per task and the

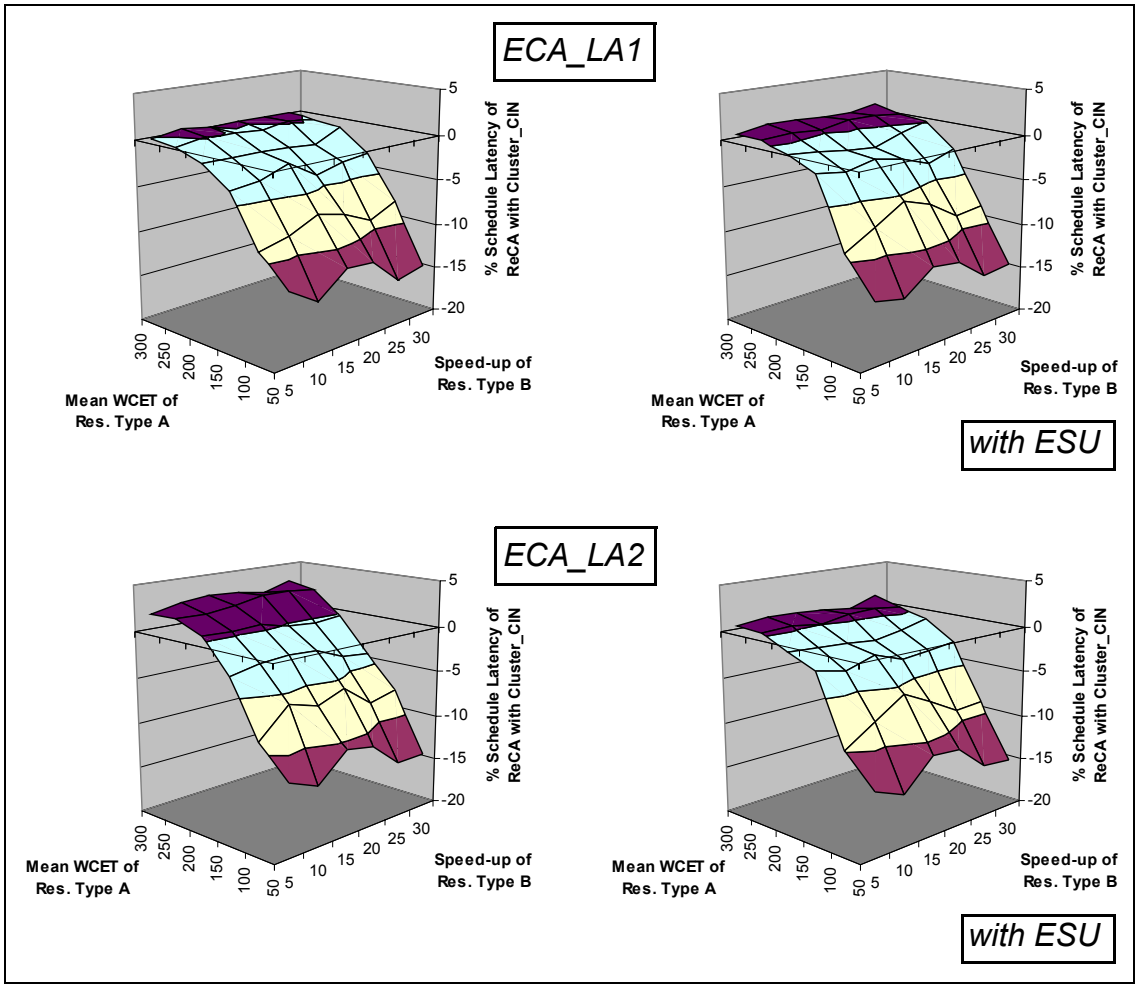


Figure 4.23: Performance Analysis of *ECA* with *Cluster_Sum* Relative to *ReCA* with *Cluster_CIN* in a Multiprocessor Scenario

sequencing of task scheduling. To improve the selection of the binding decisions, an Enhanced Constructive Algorithm (*ECA*) is introduced with an improved priority calculation including future events. Such future events are taken into account by a Look-Ahead (*LA*) parameter. *LA* is introduced to regard inevitable data transfers and to compare task/resource combinations including succeeding tasks. Three different variants of *ECA* are presented:

- *ECA_LA1* only considers mandatory data transfers.
- *ECA_LA2* selects the resources with the best performance for the immediate succeeding tasks.
- *ECA_LA3* selects the resources with the longest latency for the immediate succeeding tasks.

The analyses confirm that *ECA_LA1* performs better than *ECA_LA2* and *ECA_LA3*. *ECA_LA1* takes only known events into account for priority calculation, whereas *ECA_LA2* and *ECA_LA3* speculate about the binding information of future events. *ECA_LA2* only selects shortest latency while *ECA_LA3* allows worse resources by using the longest latency. Improvement in performance of up to 35% can be achieved by the algorithm enhancements using *LA*. However, the multitude of binding and scheduling possibilities and the dependency of the partitioning algorithms on the structure of the design model limits the predictability of the performance capabilities of *ECA*.

With the use of the Enhanced Static Urgency (*ESU*), the results of *ECA_LA2* and *ECA_LA3* can be improved in the given scenarios. However, the combination of *ECA_LA1* and *ESU* does not show significant improvements.

Clustering allows the designer to enforce Specific modeling and implementation constraints. In particular, communication events are necessary to be performed on the shared media by one implementation which allows the comparison to an alternative implementation using different resources.

Different clusters of nodes can facilitate the usage of different design objectives during partitioning. By using one task graph as analysis model, the design objective may be adapted to, for instance, parallel processing or acceleration of functionalities with the help of clustering significant nodes of the task graph.

Common Implementation Nodes (CINs) allow to differentiate between various implementation possibilities represented as resource sets. Each resource set contains a number of resources exclusively used for the tasks of a cluster, if selected. Hence, a flexible characterization with different resources for each implementation possibility is feasible.

Two extensions to cover clustering with *ECA* are introduced: *Cluster_CIN* and *Cluster_Sum*. The first extension simply ensures using the same resource set for all tasks of a cluster. The other extension considers all CINs within the cluster to adjust the priority calculation of the nodes. Using *Cluster_Sum* results in an improvement of the performance up to 15%.

The combination of *Cluster_CP* with the variants of *ECA* shows that improvements can be achieved for wide ranges of scenarios. However for large WCETs, *ECA* cannot select the most appropriate resources and can result in worse performance than *ReCA*.

In the next chapter, a real-world example is used to confirm the results obtained in this chapter. By varying the architecture, the performance of the algorithms is evaluated.

Chapter 5

Real-World Application Practice

In the previous chapter, *ECA* has been evaluated using synthetic task graphs to derive the performance of the algorithm in respect to characteristic task graph properties. TGFF has produced a huge number of synthetic CPGs with random and irregular structure representing similar characteristics. However, these graphs hardly reflect actual applications. This chapter shows the practice of *ECA* on a real-world networking application. The used networking application is an internet router application with packet forwarding functionality using DiffServ methodologies, [6] and [76].

The analysis model with CPG and target architecture is introduced. The performance of *ECA* is evaluated by applying the algorithm to different architectures. Starting with a small subset, the target architecture is extended in four stages to its full extent. Due to the missing predictability, the different target architectures are used in a real-world application practice to test the performance of the introduced algorithms.

5.1 Architecture Exploration

Today's networking applications require both characteristics high-performance and cost-efficiency. In order to identify the most suitable implementation which meets the requirements best, an extensive exploration is necessary. A by-inspection generation of the SoC architecture is not feasible due to the general complexity of real-world examples and especially communication mutual dependencies which can arise.

Concurrent communication on the SoC can cause congestions which may slow down the overall performance. Communication bottlenecks are not easy to predict, since the communication depends on the processing of the data. Therefore, a more detailed look into the communication is required during architecture exploration.

In this chapter, an architecture exploration is used for the evaluation of the partitioning algorithms by adding and changing resources of the target architecture. Figure 5.1 shows different possibilities to adapt the target architecture to different design objectives. Starting from a reference target architecture in the middle of the figure, the performance and suitability of this target architecture is evaluated. Depending on the application, HW accelerators can improve the overall performance significantly. Alternatively, applications with intensive memory accesses can require local memory for the processing unit to avoid congestions of the communication resource and an impairment of performance. Other than extending the resource sets, a DMA (direct memory access) controller can stress the architecture by generating background traffic on the shared bus to evaluate the behavior at different bus loads. This resource stores and retrieves data packets to and from the memory in networking applications without the interaction of microprocessors.

In this thesis, the influence of the number of HW accelerators and the presence of local memory on the performance of the algorithms is analyzed.

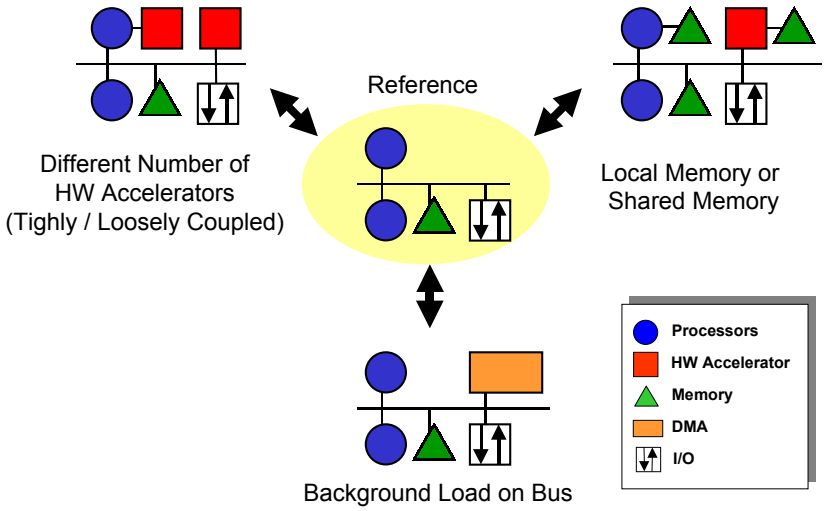


Figure 5.1: Target Architecture Evaluation

5.2 Real-World Application

To evaluate the utilizability for real-world applications, an example is taken from the networking domain. The networking application and the system environment are explained next. With this information, a CPG is derived in the following.

5.2.1 Internet Router Application Diffserv

The ingress packet processing of a router line card should serve as verification of the introduced algorithms. Thereby, the design model for this evaluation comprises the processing of the header after the storage of the packet in a data memory and before the packet is enqueued in the subsequent switching network.

For the system environment, an Ethernet network is assumed. The application comprises Ethernet switching as well as all necessary tasks to process IP packets. Beside fundamental routing functionality, *DiffServ* functionalities according to [6], [48], and [76] are included, especially the classification of the packets, policing with metering and marking, and accounting functions. In this model, only time-critical functions belonging to the "fast path" ([2]) are considered. These functions are applied to packets which are forwarded immediately. Management and control functions which are less time-critical and define "slow path" are disregarded. These tasks are usually processed on a central card due to limited performance on the line card anyway.

Figure 5.2 depicts an overview of the described application *DiffServ*. The single functionalities are shown as blocks. Some of these functionalities are particularly relevant in respect of achievable overall performance. Particularly, routing decisions with next-hop (NH) look-ups are very significant. Such a lookup determines the next router on the packet's way to the destination. Aside, packet classification can have significant impact of performance. The priority of the packet and the allocation to specific traffic streams are important features of packet classification.

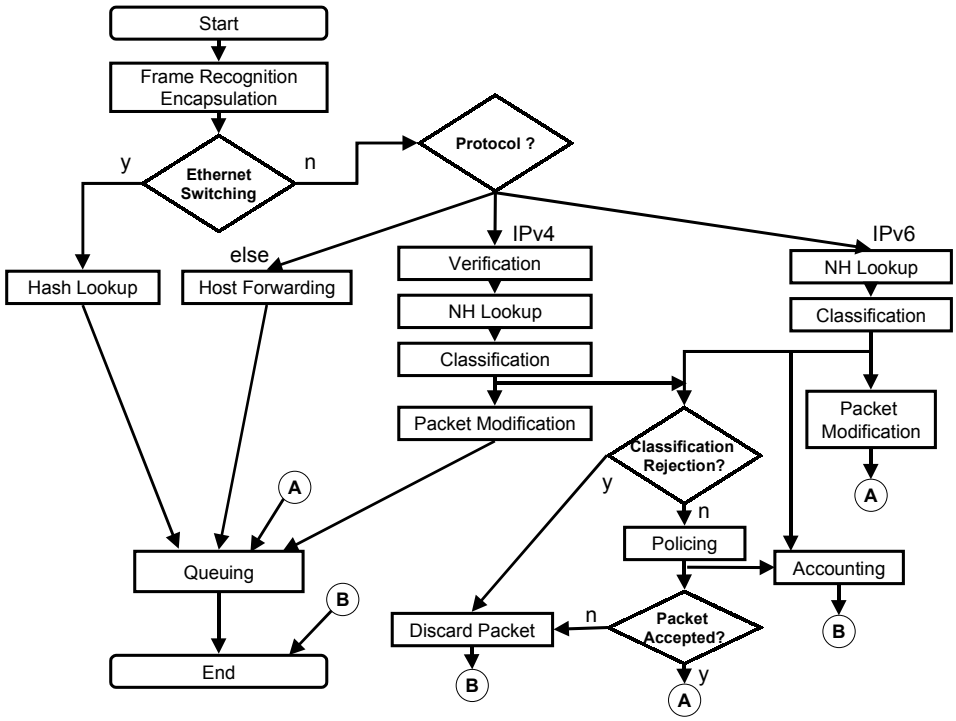


Figure 5.2: Flow Diagram of DiffServ

Routing

Within the Internet, IP packets are forward from one router to the next one in a hop-by-hop behavior. Each router determines the next leg to the next router independently by selecting the appropriate egress interface. The protocols how to determine the route are not subject of this work.

From a routing table, the nexthop lookup determines the next router along the packet's path and the dedicated router interface. With concept of CIDR (Classless Interdomain Routing), the entry which has the longest prefix match is taken for routing. In this way, the number of routing list entries can be reduced. Destination networks with consecutive address space can be treated with just one entry. It can be assumed that the longer the prefix match, the closer the destination.

The nexthop lookup is the most complex component of the IP packet processing in routers and limits the achievable performance. For this reason, this topic is subject of research regarding the time for a lookup, memory requirements, necessary preprocessing, and the possibility to modify the data base. [43] gives a broad overview of current algorithms.

For the considered example here, the nexthop lookup described in [98] is used. This algorithm is based on a binary search of hash tables in which the entries are ordered by prefix length. A nexthop lookup needs 5 memory accesses for IPv4 and 7 memory accesses for IPv6. Since the worst case situation is considered, it is assumed that each lookup needs the maximum number of accesses. The routing table is serviced by the slow path which provides, modifies and stores the routing information in the shared memory. The maintenance of this list is not considered in the following design model.

Packet Classification

A fundamental element of the DiffServ concept is the classification of incoming packets into particular traffic flows. These flows are categorized into diverse Classes of Service ([6]). Also called "per hop behavior", Class of Service guarantees the specific properties like packet loss, latency, or jitter among the service class and relative to other classes.

Classification is performed on the basis of specific header fields and results in a class identifier (Class ID) assigning the packet to a certain class of service. The first classification rule which fits header fields designates the Class-ID for that packet. Analyzed header fields for classification are source and destination address, the type-of-service field (TOS), the protocol field, and the source and destination port numbers derived from the transport layer. In addition, the number of the physical ingress interface is used in the considered algorithm.

Many algorithms were proposed for the multi-field classification. Besides hardware based solutions, e.g., with ternary CAM (Content Addressable Memory), different algorithms have been proposed in recent years. These algorithms can be differentiated in fundamental search algorithm, geometric or heuristic methods. Assessment criterions of

these algorithms are the duration of the classification process, the memory requirements, the number of supported fields, the size of the rule set, and its flexible extensibility. An overview of ongoing proposals for classification algorithms is given in [44].

For the networking example, the RFC (Recursive Flow Classification) algorithm is used that is described in [42]. With the help of complex preprocessing, this algorithm constructs a data structure in the memory which delivers the result of the classification after incremental memory accesses and linear combinations of the lookup results. The choice of parameters for the RFC algorithm needs to be balanced between the number of memory accesses and the size of memory.

Policing

For policing, it is assumed that each packet with its user (IP address) can accurately be classified in a unique Class ID which is used for forwarding. Policing parameter are defined as a leaky bucket with an average data rate and a maximum burst rate. Whenever the allowed data rate is exceeded, the packet may be discarded. The determination and modification of the policing parameters are performed by a controller in the slow path. For that reason, policing parameter updates is not considered in the application graph.

5.2.2 Design Model and Architecture Assumptions

Based on the functionality and the performance figures of the networking application just described, a design model in the form of a CPG is created. The target architecture consists of resources which may be found in networking devices. This CPG and target architecture are primarily meant to evaluate the performance of *ECA* in respect to a real-world application. In the following evaluation, the design model remains unchanged while specific target architectures are selected to emulate a potential architecture exploration.

Conditional Process Graph

The CPG of *DiffServ* consists of 152 nodes with 18 nodes describing conditional edges. This results in 12% of all nodes with conditional branches. The mean amount of processed data transferred between the nodes is 56 bits. The 103 nodes can be sped up by dedicated hardware accelerators grouped in 6 clusters. This result in a mean overall acceleration factor of 5 compared to the available processors. Appendix C shows the CPG and the target architecture used in this chapter.

Target Architectures

The target architecture of the processor types is bus-based with the following abstract resources: Two embedded processors with different properties represent a fast core and a slow core which allow all tasks to be processed. Several accelerators provide efficient processing for nexthop lookups, classification, policing, and general table lookups. The

system also includes interfaces both receiving the unprocessed packet from the framer and forwarding the processed packet to the switching network. Additionally, two independent memories hold data, one for entire packets, the other for the storage of lookup tables and other data like packet descriptors, accounting information, and policing parameters. In addition to that two independent memories, local data memory is provided to all accelerators.

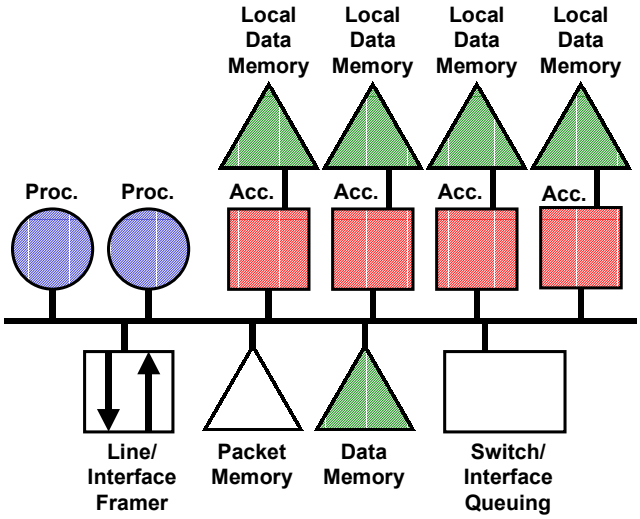


Figure 5.3: Target Architecture for DiffServ

In figure 5.3, the shaded blocks of the target architecture are considered during the analysis. The other blocks are used for the reception of the packets and their storage in memory, as well as queuing and forwarding. These functions are not taken into account in this example.

The WCET values represent the performance of the processors and HW accelerators by an upper limit of the possible execution duration. In this model, up to four accelerator blocks representing potential commercial intellectual property (IP) modules are applied in the target architecture for different functionalities: a lookup-engine, a classifier, a next-hop lookup engine and a resource manager.

5.3 Evaluation of *ECA*

The performance of the variants of *ECA* with the clustering options are compared in respect of the various target architectures. In the following, the partitioning algorithms utilizes the same CPG of *DiffServ* with five different implementation possibilities as target architectures. The numbers of HW accelerators and the usage of local memory are the distinctive features:

Scenarios	No. of Processors	Shared Memory	No. of Accelerator for	Local Memory of Acc.
DiffServ_Proc	2	x	-	-
DiffServ_1Acc	2	x	1: Classification	-
DiffServ_1AccMem	2	x	1: Classification	x
DiffServ_4Acc	2	x	4: Classification, Next-hop Lookup, Lookup, Management	-
DiffServ_4AccMem	2	x	4: Classification, Next-hop Lookup, Lookup, Management	x

Table 5.1: Different Scenarios for *DiffServ*

Each of these target architectures is applied to *ECA_LA1* and *ECA_LA2* in combination with or without *ESU*. The options *Cluster_CIN* and *Cluster_Sum* assure that the accelerators are accordingly used by the CIN. As reference, *ReCA* with *Cluster_CIN* is applied. The runtime of each algorithm never exceeds 1 second due to the size of the task graph with 152 nodes.

DiffServ_Proc

This target architecture consists of two different processors and one shared memory, see figure 5.4. All tasks can be processed by either processor with different latencies. Table 5.2 shows the results of the different versions of *ECA*.

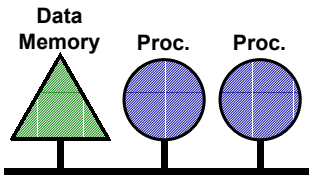


Figure 5.4: Target Architecture for *DiffServ_Proc*

It can be observed that no variant of *ECA* can achieve improvements compared to *ReCA*. Nevertheless, *ECA_LA2* performs better than *ECA_LA1*, and *ESU* can mitigate the impaired results by differentiating dynamic priority values. Also, Cluster_Sum can alleviate unfavorable selections of resources.

ECA_LA2 performs best of all variants with a schedule latency increase of 0.6%. With only two computational resources with similar performance available, *ECA_LA2* with *ESU* and the clustering option *Cluster_Sum* can select the resources in the most appropriate order with the least loss in performance.

ECA_LA1 cannot perform as in chapter 4 due to missing inevitable data transfers to fast computational resources. Since the only inevitable data transfers are to and from memory

and necessary for all compared resources in this scenario, no blocking communication resource can be identified.

% Schedule Latency of *ReCA* with *Cluster_CIN*		***Cluster_CIN***	***Cluster_Sum***
ECA_LA1 without *ESU*		+3.7%	+3.0%
ECA_LA1 with *ESU*		+3.5%	+1.9%
ECA_LA2 without *ESU*		+2.2%	+2.8%
ECA_LA2 with *ESU*		+1.7%	**+0.6%**

Table 5.2: Result of *ECA* in Scenario *Diffserv_Proc*

DiffServ_1Acc

Figure 5.4 depicts the target architecture of *DiffServ_Proc* with the extension of one application-specific accelerator for the functionalities of classification. All tasks which belong to classification may be executed on the accelerator. However, all memory accesses still needs to be performed with the shared data memory.

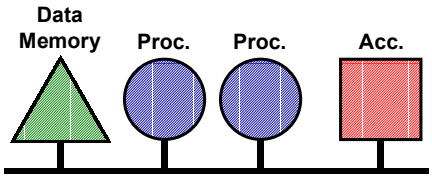


Figure 5.5: Target Architecture for *DiffServ_1Acc*

Table 5.3 shows the results evaluating the algorithm with the target architecture of *DiffServ_1Acc*. *ReCA* does not use the accelerator, because the processors are more favorable for the first CIN due to bus congestions. Although the option *Cluster_Sum* utilizes the accelerator, the latencies are longer than the one obtained by *ReCA* except for *ECA_LA1* without *ESU*. This algorithm can improve the schedule latency by 1.0%.

By not using the accelerator and avoiding communication for the data transfer to the accelerator, *ECA_LA1* without *ESU* and *Cluster_CIN* can achieve the best result in this scenario. *ECA_LA1* can utilize the processor with the slow core in more efficient way than *ECA_LA2*.

% Schedule Latency of *ReCA* with *Cluster_CIN*		***Cluster_CIN***	***Cluster_Sum***
ECA_LA1 without *ESU*		**-2.0%**	-1.0%
ECA_LA1 with *ESU*		+2.0%	+0.6%
ECA_LA2 without *ESU*		+0.3%	+0.3%
ECA_LA2 with *ESU*		+2.4%	+3.5%

Table 5.3: Result of *ECA* in Scenario *Diffserv_1Acc*

DiffServ_1AccMem

In order to analyze the results of the partitioning algorithms with respect to communications, a local data memory is appended to the accelerator from *DiffServ_1Acc*. This memory is connected directly to the accelerator on a separate connection. The data for the processing of classification is already loaded in the local memory. For the simplicity of the analysis model, the separate link has the same characteristics as the shared bus.

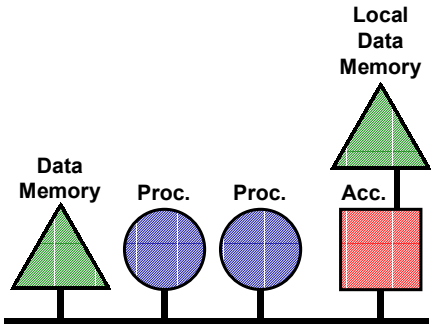


Figure 5.6: Target Architecture for *DiffServ_1AccMem*

The allocation of the accelerator is the same as in the scenario *DiffServ_1Acc*. However, the algorithms *ECA_LA1* and *ECA_LA2* with *ESU* and *Cluster_Sum* can reduce the schedule latency by 13.3% by using the faster accelerator. The separate connection to the the local memory has a utilization of 18% which relieves the shared bus and allows to achieve this performance figure.

The other variants of *ECA* do not select the accelerator for all available clusters. Nevertheless, almost all variants can achieve improvements in performance. Again, the local memory allow the shared bus to be relieved and the schedule to be shortened.

% Schedule Latency of *ReCA* with *Cluster_CIN*		***Cluster_CIN***	***Cluster_Sum***
ECA_LA1 without *ESU*		-2.2%	-4.3%
ECA_LA1 with *ESU*		-3.1%	**-13.3%**
ECA_LA2 without *ESU*		+0.1%	-3.4%
ECA_LA2 with *ESU*		-4.9%	**-13.3%**

Table 5.4: Result of *ECA* in Scenario *Diffserv_1AccMem*

DiffServ_4Acc

This target architecture is an extension of *DiffServ_1Acc* by three further accelerators: a generic lookup engine, a next-hop lookup engine, and a resource manager. All memory accesses need to be performed through the shared data memory.

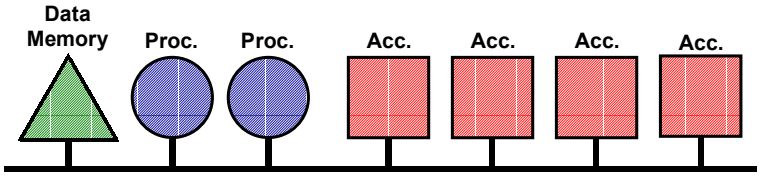


Figure 5.7: Target Architecture for *DiffServ_4Acc*

For this scenario, *ECA_LA2* without *ESU* and *Cluster_CIN* performs best with an improvement of 0.7%. By using only the generic lookup engine accelerator, similar behavior than in the scenario *DiffServ_1Acc* can be observed. The result is achieved by avoidance of communication to and from the accelerators.

ECA_LA2 without *ESU* and *Cluster_Sum* uses all four accelerators. However, the performance is the worst of the scenario. By using the accelerators, the communication latencies to and from the shared memory are not altered. The context transfers from the processors to the several accelerators are delayed which results in the observed performance.

% Schedule Latency of *ReCA* with *Cluster_CIN*		***Cluster_CIN***	***Cluster_Sum***
ECA_LA1 without *ESU*		-0.6%	+1.7%
ECA_LA1 with *ESU*		+2.9%	+2.9%
ECA_LA2 without *ESU*		**-0.7%**	+4.0%
ECA_LA2 with *ESU*		+1.7%	+3.5%

Table 5.5: Result of *ECA* in Scenario *Diffserv_4Acc*

DiffServ_4AccMem

This target architecture is based *DiffServ_4Acc* and extended by four local memories for each accelerator. Again, each local memory is connected to the accelerators via separate links having the same properties as the shared bus. For each local memory, it is assumed that the data necessary for processing is already loaded. For the coherence of the memories, the "slow path" management and control functions are responsible for updates. However, these functions are not considered in this work.

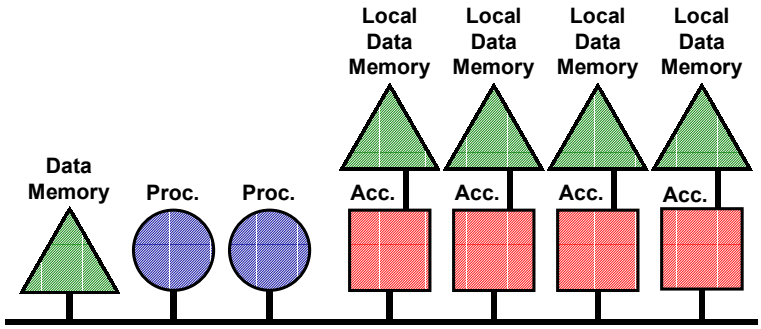


Figure 5.8: Target Architecture for *DiffServ_4AccMem*

ECA_LA1 with *ESU* and *Cluster_Sum* achieves the best result for *DiffServ_4AccMem*. This algorithm uses all accelerators and its local memories, and balances the load of the processors. In this way, it can reduce the latency by 15.1% compared to *ReCA*.

With the local memory of the accelerators, the connections to the local memory relieve the traffic load of the shared bus. *Cluster_Sum* uses the accelerators and provide about 10% shorter overall latencies than using *Cluster_CIN* by binding all accelerators.

In all cases, the usage of *ESU* improves the results. In this scenario, ambiguities of task priorities can be clarified by assigning higher priorities to tasks with more succeeding branches.

% Schedule Latency of *ReCA* with *Cluster_CIN*		***Cluster_CIN***	***Cluster_Sum***
ECA_LA1 without *ESU*		+1.6%	-13.9%
ECA_LA1 with *ESU*		-4.0%	**-15.1%**
ECA_LA2 without *ESU*		+3.1%	-9.6%
ECA_LA2 with *ESU*		-5.8%	-13.3%

Table 5.6: Result of *ECA* in Scenario *Diffserv_4AccMem*

5.4 Summary, Comments and Conclusions

In this chapter, the variants of *ECA* are applied to the design model of a networking application *DiffServ*. In comparison to the CPGs generated by TGFF, the represented functionality and the structure of this design model is based on a real-world application. The target architecture consists of two processors, one shared memory, a common bus, four accelerators. Each accelerator is equipped with local memory which is connected directly to the accelerators via a separate bus.

For the evaluation of *ECA*, the selection of target architectures is based on a potential architecture exploration. Five different subsets of the target architecture reproduce architecture instances starting with only two processors and one shared memory. By adding accelerators and optional local memory for specific functionalities, the behavior of the variants and options of *ECA* is analyzed.

For each target architecture, the best algorithms for the most schedule latency reduction is listed in table 5.7. *ECA_LA1* with *ESU* and *Cluster_Sum* can archive the best improvement with 15.1% compared to *ReCA*. The range of the observed results match the results with synthetic task graphs of chapter 4.

Scenarios		Algorithms with best performance	Reduction of Schedule Latency
DiffServ_Proc		*ECA_LA2* with *ESU* and *Cluster_Sum*	+0.6%
DiffServ_1Acc		*ECA_LA1* without *ESU* and *Cluster_CIN*	-2.0%
DiffServ_1AccMem		*ECA_LA1* with *ESU* and *Cluster_Sum,* *ECA_LA2* with *ESU* and *Cluster_Sum,*	-13.3%
DiffServ_4Acc		*ECA_LA2* without *ESU* and *Cluster_CIN*	-0.7%
DiffServ_4AccMem		*ECA_LA1* with *ESU* and *Cluster_Sum*	-15.1%

Table 5.7: Best Results of Variants of *ECA* for *DiffServ*
Compared to *ReCA* with *Cluster_CIN*

Depending on the presence of local memory in the target architecture, *Cluster_Sum* and *ESU* produces the best results by selecting all available resources to reduce the scheduling latency. In the other cases, *Cluster_CIN* without *ESU* avoids communication and achieves better results than *Cluster_Sum*.

The outcome of the algorithms is very dependent on the structure of the analysis model. Minor changes in the design model and the target architecture can result in contingent behavior and output of the algorithms. The observed results of table 5.7 also exhibit that for each scenario a different algorithm performs best. However, the short runtimes of the algorithms allow the execution of several variants and options to obtain the best solution.

Chapter 6

Summary, Conclusion and Outlook

6.1 Summary and Conclusion

In this thesis, a methodology of a constructive heuristic for partitioning of process graphs as a part of system synthesis has been introduced. This constructive heuristic utilizes future processing steps of the application for resource selection and scheduling. Communication is intensively taken into account for priority calculation to avoid bottlenecks on shared buses. The modeling of networking applications necessitates the handling of conditional branches to include various networking protocols in one design model. The support of mutual exclusion of tasks within the design models allows the representation of the worst-case scenario without applying all alternatives of conditional branches. Thus, this partitioning algorithm is suitable for the design of networking applications.

The performance of constructive heuristics depends on the priority calculation of the list scheduling algorithm. These priorities account for two factors: the resource selection and the sequencing of task scheduling. By using additional future processing steps of the application for the priority calculation, the performance of the algorithms could be improved. The consideration of such events allows the selection of more appropriate resources and more favorable scheduling of the tasks.

Clustering of Common Implementation Nodes (CIN) can enforce special implementation provisions during partitioning. For instance, the evaluation of memory accesses of interrelated behavioral objects require to use the same resource. Moreover, resource sets may extend the implementation possibilities to several components instead of a single resource in a common model.

Various partitioning approaches are available to refine high level design models towards lower levels of abstraction. Exact methodologies can provide optimum solutions for small problems. However, the determination of optimum solutions requires long runtimes of the algorithms which limits the problem to small sizes. With increasing problem sizes,

heuristics can determine solutions in a practicable timeframe. By approximating the optimal solution, the results of heuristic algorithms may be sufficient to meet the application requirements. Compared to iterative heuristics, constructive heuristics allow very short execution time to determine solutions.

For the considered application domain networking, a partitioning algorithm had to be selected which can handle large design models. Moreover, fast runtimes, the support of conditional branches, the consideration of communication during partitioning, and the support of heterogeneous target architectures are prerequisites. Figure 6.1 gives an overview of the characteristics of the potential partitioning heuristics introduced in chapter 2 and the introduced algorithms of this thesis.

Partitioning Algorithms	Dave et al., [22], [23] "COSYN", "COHRA"	Eles et al., [31], [32], [33] "Cond.Proc.Graph	Kalavade et al., [57] "Dyn.Level / GCLP	Kwok et al., [61] "Dynamic CP Scheduling"	Sih et al., [85] "Compile-time Scheduling"	Vallerio et al., [94] "Graph Transformations"	Wild, [101] "Enhanced CP Scheduling"	Wu et al., [105] "Sched.&Mapp. Of CTG"	Xie et al., [107] "Alloc. & Sched. Of CTG"	Zitzler et al., [109] "Evolutionary Algorithms"	Brunnbauer "ReCA"	Brunnbauer "ECA"
Iterative	✓	✓				✓	✓	✓		✓		
Constructive			✓	✓	✓				✓		✓	✓
Dynamic List Scheduling		✓	✓	✓	✓		✓	✓	✓		✓	✓
Look-Ahead		✓	✓	✓								✓
Conditional Branches		✓					✓	✓	✓		✓	✓
Considering Comm. Latencies	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Considering Memory Latencies							✓			✓	✓	✓
Hierarchy, Clustering	✓					✓						✓
Multiprocessorsystems	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Appl. Spec. HW Accelerators	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓
Flexible Comm. Architecture		✓	✓		✓			✓		✓	✓	✓

Figure 6.1: Overview of Heuristic Algorithms for Partitioning

The constructive heuristic of Xie et al., [107], complies with the requirements above and has been selected as the basis of the considered partitioning algorithm in this thesis. The algorithm processes design models in the form of conditional process graphs (CPG) and utilizes estimations of worst-case execution times (WCET) which capture the properties of the target architectures. In this thesis, the algorithm of Xie et al. has been enhanced to allow synthetic CPG in combination with conditional branches to be used for evaluation. The resulting Reference Constructive Algorithm (*ReCA*) contains no different treatment of resources, variable length of data transfers within a more flexible communication structure, hole filling of empty slots during scheduling for low latency scheduling, and a generalization of conditional branches for arbitrary connections within the CPG.

The evaluation of chapter 3 has shown that *ReCA* performs much faster than iterative algorithms. However, *ReCA* cannot achieve the performance of this type of algorithms. To compensate for this performance impairment, the Enhanced Constructive Algorithm (*ECA*) has been introduced. Its improved priority calculation includes inevitable future events. In

this thesis, three versions of Look-Ahead (*LA*) for immediate succeeding events have been introduced. *ECA_LA1* only takes inevitable future data transfers into account. *ECA_LA2* also includes the fastest succeeding tasks, while *ECA_LA3* considers the slowest performing tasks which succeeds immediately. In addition, the calculation of Static Urgency (*SU*) has been modified to improve the sequencing of the tasks scheduling by Enhance Static Urgency (*ESU*).

With clustering, a group of nodes or functionalities, here so-called CINs (Common Implementation Nodes), can be constrained to specific implementation possibilities. In this way, different levels of granularity can be used which facilitates the model generation. In addition, the same task graph can serve for different design objectives with various implementation possibilities. For instance, exploiting parallelism with multi-processor systems, or improving the performance of single functionalities by application-specific hardware accelerators may be the focus of the architecture evaluation.

The performance has been evaluated with both synthetic design models generated by TGFF, [24], and a design model of a real-world example *DiffServ*, a networking application on a router linecard. The mandatory support of conditional branches in a design model limits the number of partitioning algorithms which allow the processing of synthetic task graphs.

The results show that the algorithmic improvements of *ECA* can provide significant better solutions compared to *ReCA* by looking ahead the potential schedule and utilizing future events for priority calculation. However, some parameter sets cause *ReCA* to perform better than *ECA*. Among the variants of *ECA*, *ECA_LA1* performs best. The known data transfer events allow the precise estimation of the subsequent latencies and allow to avoid communication bottlenecks. *ECA_LA2* cannot achieve the same performance as *ECA_LA1*, since assumptions of the succeeding events are used for prediction. The variety of binding and scheduling possibilities of the succeeding tasks complicate the prediction of the actual assignment. *ECA_LA3* uses the longest latency for the immediate succeeding data transfers and tasks as *LA* value which causes the selection of the slowest resource for the considered task. In this way, data transfers and potential communication bottlenecks can be avoided. The overall performance of *ECA_LA3* exhibited much lower performance than the other two variants mostly caused by the selection of slow resources. Hence, only *ECA_LA1* and *ECA_LA2* have been considered for further investigation.

ESU can resolve task priority ambiguities for *ECA_LA2* and *ECA_LA3* and lead to better results. It takes the structure of the process graphs into account compared to the critical path through the process graph. Since the *LA* value of *ECA_LA1* considers only inevitable data transfers and no succeeding tasks, *ESU* does not show significant influence on *ECA_LA1* in the considered scenarios with synthetic task graphs.

Clustering with the algorithm *Cluster_Sum* leads to good results in most cases. The usage of all CINs within a cluster for the priority computation is more favorable instead of only the first CIN with the algorithm *Cluster_CIN*. Depending on the WCET distribution of the corresponding CIN in the cluster, *Cluster_CIN* can achieve similar performance if the WCET of the first CIN represents the latency characteristic of the entire cluster.

A significant drawback of the introduced algorithm is the dependency on the structure of the design model and the properties of the target architecture. The variants of *ECA* have shown improvements in the mean over a huge number of design models and target architectures. However, the results of *DiffServ* have shown that each variant of the algorithms performs different on the diverse target architectures.

For various design models and target architectures, each variant behaves differently and needs to be evaluated individually. The performance of the algorithmic improvements of *ECA* cannot be predicted. However, the very fast runtimes of *ReCA* and *ECA* allow the processing of large task graphs and the subsequent usage of different variants for the same scenario without noticeable delays.

6.2 Outlook

In this thesis, a new constructive heuristic for partitioning has been introduced which intensively takes communication into account to avoid communication bottlenecks. The algorithms exclusively utilizes latencies for the performance optimization. The consideration of other criteria, such area, cost, and power of the used resources, requires more complex partitioning algorithms with multidimensional optimization. The introduced constructive algorithms of this thesis may provide an initial solution to such extended algorithms.

In the introduced real-world practice, one single data packet was processed by *DiffServ*. For the evaluation of timely overlapping data packets, a design model with an accordant number of independent instances of the introduced CPG of *DiffServ* has to be applied. To handle such design models, *ECA* needs to be extended to process multiple data packets accordingly. As a first step, the replication of several CPGs can be merged into a new CPG with according delays representing the arrival times. The processing of such a new CPG requires the correct assignment of task priorities to ensure the appropriate sequence of parallel CPG instances.

The provision of methods and tools for the creation and evaluation of suitable VLSI architectures is a crucial task to reduce the productivity gap, [51]. This thesis provides a contribution to system synthesis with a fast partitioning algorithm supporting applications with control dependencies. Many challenges still lie ahead to rise the entry level of chip design to higher levels than today by providing suitable exploration and synthesis tools.

Appendix A

Tools

A.1 Environment for Partitioning Algorithm Evaluation

Figure A.1 gives an overview of the tool used for the analysis of *ReCA* and *ECA* whose usage is described in the following:

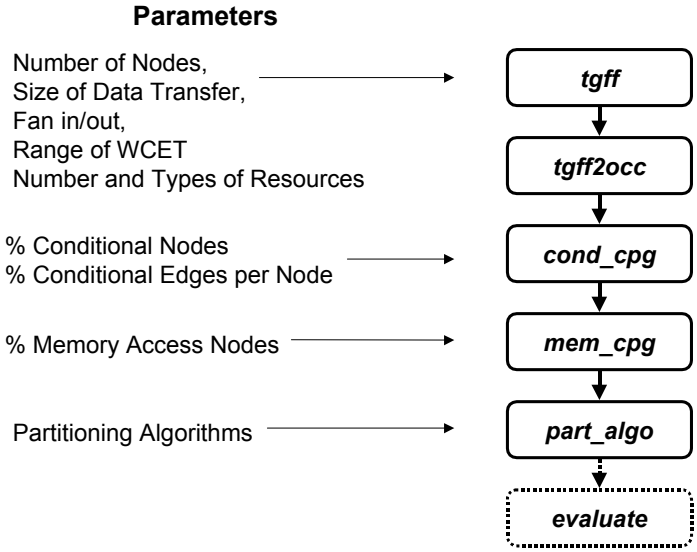


Figure A.1: Tool Chain

tgff

Task Graphs For Free (TGFF), [24], is designed to provide a flexible and reproducible way of generating pseudo-random task-graphs for use in scheduling and allocation research. This includes the areas of embedded systems, hardware/software co-design, parallel or distributed hardware or software studies, as well as any other area which requires problem instances consisting of directed acyclic graphs (DAGs) of tasks, i.e., task-graphs. It also generates associated resource parameters in accordance with the user's parameterized graph and database specifications and thereby allows the generation of graphs tailored to particular domains.

usage: tgff file[.tgffopt]

In the following, an excerpt of the instructions of TGFF used in this work is given. The file is a command file with .tgffopt extension describing a scenario with a CPG with 40 nodes and a target architecture with two processors, two accelerators and a shared bus. The transfer and task types are used to randomized the values in the WCET tables:

```
# Graph Data                    // Comment
tg_label Graph                  // Name of the task graph, here "Graph"
tg_cnt 1                        // Number of graphs generated
period_mul 1                    // Multiplier for periods in multirate systems
task_cnt 40 5                   // Number nodes: 40 + 5 nodes
task_degree 5 5                 // Maximum number of transmits
                                // (fan in, fan out) per task, here up to 5
                                // connections on the ingress and egress side
task_type_cnt 100               // Number of different task types
tg_write                        // Write the task graphs to .tgff file
# Transfer
trans_type_cnt 100              // Number of transmit types
table_label Data                // Label used for tables, here "Data"
table_cnt 1                     // Number of tables, here 1 bus.
type_attrib bits 51.0 50.0 1.0 1.0  // Name, average, jitter, multiplier,
                                // Round to (0.0 means no rounding)
                                // Transfers: 51 ± 50 bits
trans_write                     // Write transmission event information
# Processors
table_label Res                 // Label used for tables, here "Res"
table_cnt 2                     // Number of tables, here 2 processors.
type_attrib feasibility 20.0 20.0 1.0 1.0, WCET 80.0 30.0 1.0 1.0
                                // Feasibility: 20 ± 20 (100% of all tasks)
                                // WCET: 80 ± 30 time units
pe_write                        // write processing elements information
```

```
# ASICs
table_label Res                    // Label used for tables, here "Res"
table_cnt 2                        // Number of tables, here 2 ASICs.
type_attrib machbar -10.0 20.0 1.0 1.0, dauer 20.0 16.0 1.0 1.0
                                   // Feasibility: -10 ± 20 (25% of all tasks)
                                   // WCET: 20 ± 16 time units
pe_write                           // write processing elements information
eps_write                          // write task graph as PostScript
```

The parameter *Feasibility* is used by *tgff2occ* in the next step. The random number identifies the mean availability of the considered resouce for the execution of the tasks within the CPG. A non-negative value indicates the task to be available.

tgff2occ

This graph format conversion tool converts the output of TGFF to another format used by the constructive algorithm tool *eca*.

usage: tgff2occ file[.tgff]

The output of tgff2occ consists of a file describing the CPG, the resources and the communication structure:

-example.cpg (Information of Conditional Process Graph)
<Number of Nodes in the CPG>
  <Number of Predecessors of considered Node>
    <Node Number Predecessors> <Coming from a Conditional Branch?>
      :
  <Number of Successors of Node of the considered Node>
    <Node Number Successor> <Bits to transfer> <Conditional>
      :
  :

-example.res (WCET of Resources for all Nodes)
<Number of Resources>
  <WCET for Res1>:<WCET for Res2>:... // For each considered node
  : (for all Nodes)

-example.com (Description of Communication Architecture)
<Number of Busses>
  <Bandwidth in bits> <> <> <> // Additional Info not considered yet
  <Number of Resources writing on the bus> - *<Res1> <Res2> ...*
  <Number of Resources reading on the bus> - *<Res1> <Res2> ...*
  :

cond_cpg

With the help of this tool, branches within a task graph can be assigned as conditional branches. Out of all nodes of the CPG, the percentage *cond_node_share* of nodes are selected to originate conditional branches. Out of each selected node, the percentage *cond_edge_share* of all succeeding edges is selected to actually become conditional. The rest of the edges remain unconditional.

usage: *cond_cpg name[.cpg] cond_node_share [int %] *
cond_edge_share [int %] seed

mem_cpg

This tool inserts memory nodes in the existing task graph. In this work, only one type of memory is used. A memory resource is append to the target architecture as a further resource.

usage: *mem_cpg name[.cpg] no_of_memory_nodes seed*

part_algo: eca

The tool *eca* represent the implementation of the *ReCA* and *ECA* algorithms introduced in chapter 3 and 4 using in the tool chain.

usage: *eca -[options] <sample>*

With the following options used in this work:

1..3 - Variants of Look-Ahead (default = 0)
a - omit analysis; printing just the results
A - print resource allocation
v - Verbose; printing all kinds of messages during processing
p - progress; printing progress of sub function
P - printing critical path
S - Using Enhanced *SU* (*ESU*); *SU* sum of all branches
G - Turn off all cluster *DU* calculation
g - No consideration of clustering
i - ignore Communication-Architecture (only one bus); no *.com necessary

evaluate

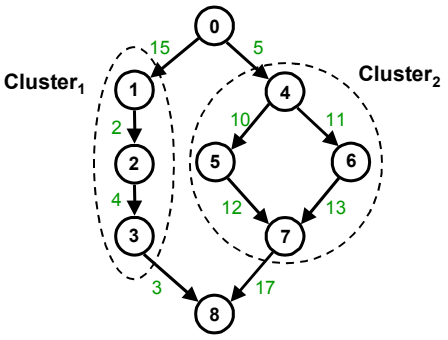
The outcome of the algorithms will be summarized. The output figures are the average improvement of the corresponding version of *ECA* compared to *ReCA* in percent, the ratio of improvements, and the ratio of deteriorations of performance.

usage: evaluate_results relevance_limit (in percent, int) in_value (int) \ n_variants n_resources

The parameter relevance_limit allows to extract results exceeding <relevance_limit> % of the reference algorithm. Tese task graphs can be examined later on. The parameter in_value is used in the output to differentiate between different algorithm evaluated by the tool. n_variants and n_resources are necessary parameters for the formatting of the result data.

A.2 Simulation Environment for Clustering

For the generation of CPGs suitable to analyze clustering, a modified tool chain is necessary. The generation of task graphs with self-contained functionalities represented as clusters of tasks requires a multi-stage procedure. Instead of generating the task graph at once, smaller self-contained functionalities task graphs are created and inserted in the main task graph, [60], see figure A.2. In this way, the refinement of single functionalities with more detail can be represented.



WCET Table

Node	Proc 1	Proc 2	Acc 1	Acc 2	
0	**2**	**10**	-	-	
1	**20**	**80**	**6**	-	
2	**30**	**90**	**10**	-	Cluster$_1$
3	**40**	**100**	**13**	-	
4	**50**	**110**	-	**12**	
5	**60**	**120**	-	**15**	
6	**70**	**130**	-	**17**	Cluster$_2$
7	**80**	**140**	-	**20**	
8	**5**	**20**	-	-	

Figure A.2: Desired Task Graph Shape for Clustering

According to the tool chain for *ECA* in figure A.1, figure A.3 depicts the interworking of the single scripts and tools to create an appropriate CPG. *create_csv* is the main

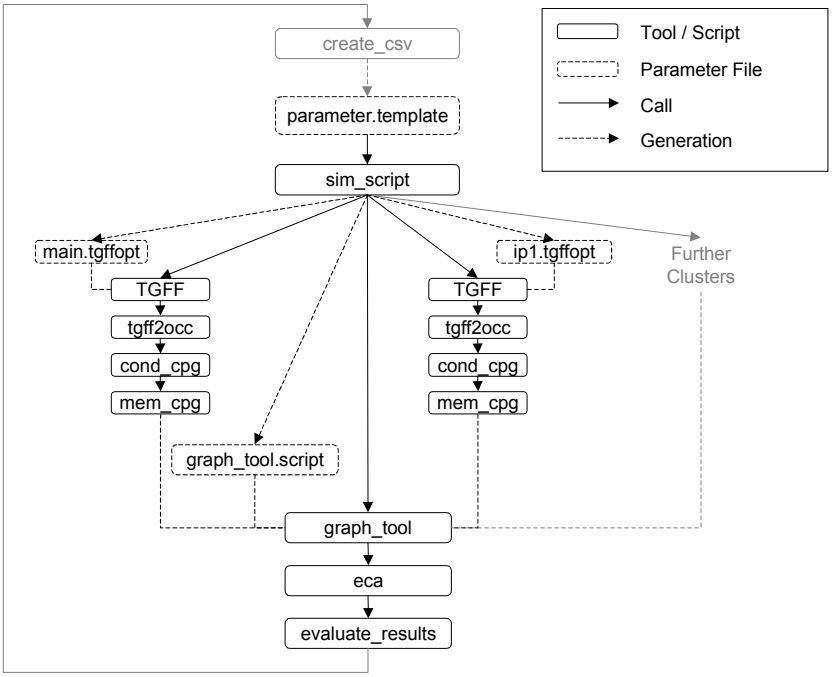


Figure A.3: Script Flow for Clustering

simulation control script which is also responsible for the formatting of the results in the csv (comma separated values) format.

parameter.template contains all parameters of the CPG and the target architecture generation for each sub CPG. This script calls the script *sim_script* which performs the task graph generation similar to the CPG generation for *ECA*.

The graph modification tool *graph_tool*, [60], combines all generated CPGs to one large task graph. The process of merging the CPGs can be automated by scripts. The created final CPG is then provided to *eca*. The results of *eca* are gathered and sorted by *create_csv*. This output can be charted by a spread sheet program, such as Microsoft Excel.

In the following, the commands of *graph_tool* are presented with an example how a final CPG is created.

Commands of graph_tool	Remark
c g	create new graph\n
c n <count>	create nodes
c e <start_node> <end_node> <transfer>	create edges
c r <count>	create resources
c i <orig.resource> <factor>	create ip-module
r n <#>	remove node <#>
r e <start_node> <end_node>	remove edge
r r <#>	remove resource <#>
d n <#> <count>	duplicate node <#>
d g {<#>,<#>, ... ,<#>} <count>	duplicate group of nodes
d r <resource> <count>	duplicate resource
e r <#> {<wcet>:<wcet>: ... :<wcet>}	edit resource at node <#>
e e <start_node> <end_node> u||c||<transfer>	edit edge
e s <seed>	edit seed
i g <#> <name>	insert graph at node <#>
i r <number>	insert resource
n	normalize
s n <#>	show node <#>
s m	show matrix
s r	show resources
s g	show groups
s a <file name>	pipe all 'shows' in file
l <graph name>	load graph
w [graph name]	write graph
x <script filename>	execute script file
h	help
q	quit

Appendix B

Performance Figures of *ECA*

The evaluation results produced in this thesis are tabularly presented in this appendix. The following results are determined with eq. (6.1). Each working point is computed by *n* results of the considered algorithm and set in to relation with *ReCA*.

$$Reduction\ in\ Latency = \frac{1}{n} \sum_{N} \left(\frac{Latency(Algorithm) - Latency(ReCA)}{Latency(ReCA)} \right) \qquad (6.1)$$

The following tables show the results of the corresponding algorithms in relation to the mean WCET of processors and the mean latency of data transfers as reduction in schedule latency compared to *ReCA*.

B.1 Results of *ECA* Applied to Synthetic Design Models

		Mean Latency of Data Tranfers					
		20	**30**	**40**	**50**	**60**	**70**
Mean Latency of Resources Type A	**50**	-8,6%	-19,8%	-24,8%	-28,3%	-27,3%	-29,5%
	100	-3,8%	-12,6%	-19,4%	-22,3%	-26,1%	-29,6%
	150	-0,9%	-8,3%	-14,4%	-17,7%	-23,3%	-25,9%
	200	-1,0%	-5,8%	-11,7%	-16,0%	-19,9%	-21,3%
	250	+0,1%	-3,6%	-8,1%	-12,3%	-15,8%	-19,1%
	300	+0,4%	-2,1%	-5,3%	-10,1%	-14,8%	-16,4%

Table B.1: Performance of *ECA_LA1* Compared to *ReCA* Considering Inevitable Data Transfers (Figure 4.18)

		Speed-up of Resources Type B					
		5	**10**	**15**	**20**	**25**	**30**
Mean Latency of Resources Type A	**50**	-31.0% / -22.6% / -5.7%	-29.3% / -20.7% / -6.8%	-32.8% / -22.4% / -4.5%	-33.9% / -22.7% / -6.5%	-32.7% / -22.4% / -3.5%	-31.8% / -24.6% / .-2.7%
	100	-22.5% / -11.9% / -2.1%	-22.4% / -11.4% / -2.4%	-22.7% / -11.0% / -1.4%	-20.9% / -10.4% / -1.7%	-21.5% / -9.9% / -2.0%	-23.6% / -11.2% / -2.7%
	150	-16.3% / -7.4% / -0.6%	-14.8% / -5.7% / -0.1%	-14.7% / -5.9% / 0.0%	-14.7% / -6.2% / +0.2%	-14.8% / -6.0% / 0%	-15.2% / -6.4% / +0.8%
	200	-10.3% / -4.1% / +1.1%	-10.5% / -3.5% / +1.0%	-9.8% / -3.6% / +1.2%	-9.4% / -3.9% / +1.4%	-9.1% / -3.9% / +1.2%	-10.1% / -4.2% / +0.7%
	250	-8.0% / -3.6% / +0.8%	-7.4% / -2.4% / +2.2%	-8.5% / -3.0% / +1.0%	-7.6% / -3.3% / +1.3%	-6.5% / -2.3% / +1.2%	-7.3% / -3.0% / +2.2%
	300	-5.6% / -2.9% / +1.4%	-6.0% / -2.4% / +1,7%	-4.9% / -1.8% / +1.8%	-5.7% / -2.6% / +1.4%	-4.5% / -2.1% / +1.2%	-5.0% / -2.2% / +1.6%

Table B.2: Performance of (*ECA_LA1* / *ECA_LA2* / *ECA_LA3*) without *ESU* for the Multiprocessor Scenario in Figure 4.19 Relative to *ReCA*

		Speed-up of Resources Type B					
		5	**10**	**15**	**20**	**25**	**30**
Mean Latency of Resources Type A	**50**	-30.4% / -23.8% / -7.0%	-30.7% / -21.5% / -6.4%	-33.8% / -23.1% / -6.0%	-33.4% / -24.9% / -7.1%	-33.3% / -23.0% / -4.3%	-30.6% / -24.6% / .-4.6%
	100	-22.6% / -12.0% / -3.8%	-22.9% / -13.0% / -3.7%	-22.8% / -12.2% / -3.7%	-21.3% / -11.2% / -3.1%	-22.5% / -11.6% / -3.5%	-22.8% / -12.1% / -3.6%
	150	-16.1% / -8.7% / -2.3%	-14.7% / -7.3% / -2.6%	-14.7% / -7.6% / -1.8%	-15.7% / -7.7% / -2.8%	-14.9% / -7.0% / -1.5%	-15.3% / -7.5% / -1.7%
	200	-10.3% / -5.4% / -0.9%	-10.4% / -5.6% / -1.2%	-9.9% / -5.2% / -1.1%	-9.6% / -4.8% / -0.7%	-9.7% / -5.2% / -0.6%	-10.3% / -5.2% / -1.4%
	250	-8.0% / -5.0% / -1.5%	-6.6% / -3.5% / -1.2%	-7.7% / -4.8% / -0.7%	-7.8% / -4.9% / -1.0%	-6.4% / -3.8% / -0.4%	-7.4% / -4.5% / -0.6%
	300	-4.7% / -3.9% / -0.8%	-5.2% / -3.3% / -0,7%	-4.7% / -3.6% / -0.2%	-5.3% / -3.8% / -1.0%	-4.4% / -3.1% / -0.3%	-5.7% / -3.3% / -0.2%

Table B.3: Performance of (*ECA_LA1* / *ECA_LA2* / *ECA_LA3*) with *ESU* for the Multiprocessor Scenario in Figure 4.19 Relative to *ReCA*

B.2 Results of Clustering Applied to Synthetic Design Models

		Mean Data Transfer Latency					
		20	**30**	**40**	**50**	**60**	**70**
Mean Latency of Resources Type A	**50**	-13.6%	-15.2%	-11.3%	-11.9%	-14.3%	-13.9%
	100	-7.3%	-4.9%	-4.7%	-5.1%	-6.7%	-5.1%
	150	-2.3%	-1.1%	-1.2%	-1.7%	-1.5%	-1.2%
	200	-0.5%	-0.8%	-0.4%	-1.0%	-0.9%	-0.4%
	250	-0.3%	+0.2%	-0.1%	-0.1%	-0.2%	-0.1%
	300	-0.3%	-0.1%	-0.2%	-0.0%	-0.1%	+0.1%

Table B.4: Multiprocessor System with *ReCA* with *Cluster_Sum* of Figure 4.21

		Speed-up of Resources Type B					
		5	**10**	**15**	**20**	**25**	**30**
Mean Latency of Resources Type A	**50**	+28.9% / +29.4%	+27.3% / +26.6%	+30.5% / +29.9%	+27.4% / +26.9%	+30.8% / +31.4%	+27.7% / +26.9%
	100	+11.6% / +13.5%	+13.3% / +13.7%	+11.2% / +13.3%	+11.7% / +13.6%	+12.7% / +14.5%	+13.7% / +14.7%
	150	+5.4% / +8.2%	+5.8% / +7.5%	+5.1% / +7.9%	+5.2% / +7.6%%	+4.7% / +7.3%	+5.2% / +7.5%
	200	+2.7% / +4.7%	+2.4% / +5.0%	+2.8% / +4.9%	+1.3% / +3.8%	+1.6% / +4.3%	+2.0% / +4.0%
	250	+1.2% / +3.4%	+0.7% / +3.1%	+0.4% / +2.9%	+0.4% / +3.5%	+0.3% / +3.1%	+0.3% / +3.1%
	300	+0.5% / +2.8%	+0.2% / +2.3%	+0.1% / +2.3%	+0.3% / +2.6%	+0.1% / +2.8%	+0.4% / +2.8%

Table B.5: Performance of (*ECA_LA1* / *ECA_LA2* / *ECA_LA3*) and *Cluster_CIN* without *ESU* of Figure 4.22

		Speed-up of Resources Type B					
		5	**10**	**15**	**20**	**25**	**30**
Mean Latency of Resources Type A	**50**	+24.8% / +25.0%	+24.1% / +24.1%	+28.8% / +29.0%	+24.6% / +24.7%	+30.3% / +29.4%	+27.4% / +26.5%
	100	+13.8% / +12.7%	+13.0% / +12.4%	+12.3% / +12.0%	+12.7% / +11.6%	+13.8% / +12.4%	+14.0% / +13.7%
	150	+6.5% / +6.1%	+6.3% / +6.2%	+6.2% / +6.1%	+6.1% / +5.4%	+6.0% / +5.8%	+6.2% / +6.0%
	200	+3.5% / +3.2%	+3.3% / +2.9%	+4.0% / +3.6%	+2.4% / +2.3%	+2.6% / +2.3%	+2.5% / +2.1%
	250	+1.6% / +1.5%	+1.2% / +1.1%	+1.1% / +0.9%	+1.3% / +1.1%	+1.3% / +1.1%	+0.7% / +0.8%
	300	+1.1% / +0.9%	+1.0% / +0.8%	+0.5% / +0.5%	+0.7% / +0.7%	+1.0% / +0.7%	+0.8% / +0.7%

Table B.6: Performance of (*ECA_LA1* / *ECA_LA2* / *ECA_LA3*) and *Cluster_CIN* with *ESU* of Figure 4.22

		Speed-up of Resources Type B					
		5	**10**	**15**	**20**	**25**	**30**
Mean Latency of Resources Type A	**50**	-12.5% / -12.1%	-14.2% / -13.5%	-11.7% / -10.7%	-12.0% / -11.2%	-15.0% / -14.0%	-14.3% / -14.0%
	100	-9.1% / -9.0%	-8.3% / -5.9%	-6.8% / -6.8%	-7.7% / -5.7%	-9.1% / -8.2%	-7.8% / -7.4%
	150	-3.7% / -3.0%	-3.3% / -2.4%	-2.1% / -2.2%	-4.1% / -2.4%%	-3.3% / -3.6%	-3.0% / -4.0%
	200	-1.4% / +0.5%	-1.5% / +0.3%	-1.6% / +0.5%	-1.6% / -0.2%	-2.0% / -0.4%	-0.5% / -0.2%
	250	-0.4% / +1.8%	+0.1% / +1.7%	-0.1% / +2.1%	-0.3% / +2.6%	+0.2% / +2.6%	-0.2% / +2.4%
	300	-0.1% / +1.7%	+0.4% / +2.4%	0.0% / +2.7%	+0.2% / +2.5%	+0.1% / +2.1%	+0.2% / +2.9%

Table B.7: Performance of (*ECA_LA1* / *ECA_LA2* / *ECA_LA3*) and *Cluster_Sum* without *ESU* of Figure 4.23

		Speed-up of Resources Type B					
		5	**10**	**15**	**20**	**25**	**30**
Mean Latency of Resources Type A	**50**	-13.2% / -13.0%	-13.9% / -14.4%	-11.5% / -11.2%	-12.0% / -11.6%	-14.5% / -14.3%	-14.2% / -14.7%
	100	-9.9% / -9.9%	-7.4% / -7.7%	-5.5% / -5.6%	-6.6% / -7.2%	-8.7% / -8.9%	-8.3% / -8.8%
	150	-1.9% / -2.5%	-2.4% / -3.1%	-1.6% / -2.2%	-2.4% / -3.3%%	-3.5% / -3.0%	-2.4% / -2.6%
	200	-0.6% / -1.1%	-0.9% / -1.4%	-1.3% / -1.5%	-1.6% / -1.8%	-0.7% / -1.3%	+0.1% / -0.6%
	250	+0.3% / +0.2%	+0.5% / +0.3%	+0.2% / 0.0%	+0.9% / +0.6%	+0.9% / +0.7%	+0.5% / -0.3%
	300	+0.4% / +0.3%	+0.9% / +0.7%	+0.8% / +0.7%	+0.7% / +0.5%	+0.6% / +0.2%	+1.2% / +1.1%

Table B.8: Performance of (*ECA_LA1* / *ECA_LA2* / *ECA_LA3*) and *Cluster_Sum* with *ESU* of Figure 4.23

B.3 Results of Real-World Application

In this section, the task graph of *DiffServ* and the according WCET table is introduced. This model acts as analysis model for the introduced algorithms of this thesis representing a real-world application.

Design Model

The model used for the real-world application analysis is shown in this chapter. The CPG of *DiffServ* can be found in figure B.1. It consists of 152 nodes and 18 nodes originate conditional branches.

In table B.9, the resource implementation library is depicted for each node in the CPG. Along with the WCET, a brief description of the functionality is given.

Number	Description	Processor 1	Processor 2	Shared Memory	Lookup-Engine	Lookup-Engine Memory	Classifier	Classifier Memory	Next-hop Lookup	Next-hop Lookup Memory	Resource Manager	Resource Manager Memory
0	Start Node	1										
1	Memory			10								
2	Recognize Frame	4	6									
3	Type	6	8									
4	LLC PDU Differentiation	8	10									
5	Switching Decision	6	8									
6	Dummy Task	1	1									
7	VLAN Tag set	3	4									
8	Unicast	2	2									
9	Prepare Lookup Hash etc.	15	20		1							
10	Lookup			4		1						
11	MAC Address	2	2		1							
12	ARP Lookup	20	25		1							
13	Memory Access			1		1						
14	MAC Address in ARP Cache	2	2		1							
15	ARP Procedure	50	50									
16	Store MAC Address	2	2									
17	Storing			8								
18	Update Packet Descriptor	4	4									
19	Enqueue Packet Address for Free-List	6	6									
20	Fetch Address for Free-List	2	2									
21	Lookup			4								
22	Descriptor enlist current packet in Free-List	4	4									
23	Storing			8								
24	Header Check	6	6									
25	IP Header Correct	22	30									
26	Check Options	3	4									
27	Decrement TTL	3	4									
28	Examine Dest-Address	5	5									
29	Extract Protocol Header	6	6									
30	Flow ID Calculation	10	20									
31	Create Chunks	18	27				1					
32	Lookup			2				1				
33	Lookup			2				1				
34	Lookup			2				1				
35	Lookup			2				1				
36	Lookup			2				1				
37	Lookup			2				1				
38	Lookup			2				1				
39	Lookup			2				1				
40	Lookup			2				1				
41	Create New Chunks	9	12				1					
42	Create New Chunks	9	12				1					
43	Create New Chunks	9	12				1					
44	Lookup			2				1				
45	Lookup			2				1				
46	Lookup			2				1				
47	Create New Chunks	9	12				1					
48	Lookup			2				1				
49	Dummy	2	2				1					
50	Packet Memory Access			4								
51	Update IP Packet	30	45									
52	16 bit Prefix Generation	4	4						1			
53	Lookup			2						1		

Table B.9: WCET Table of the Resources for *DiffServ*

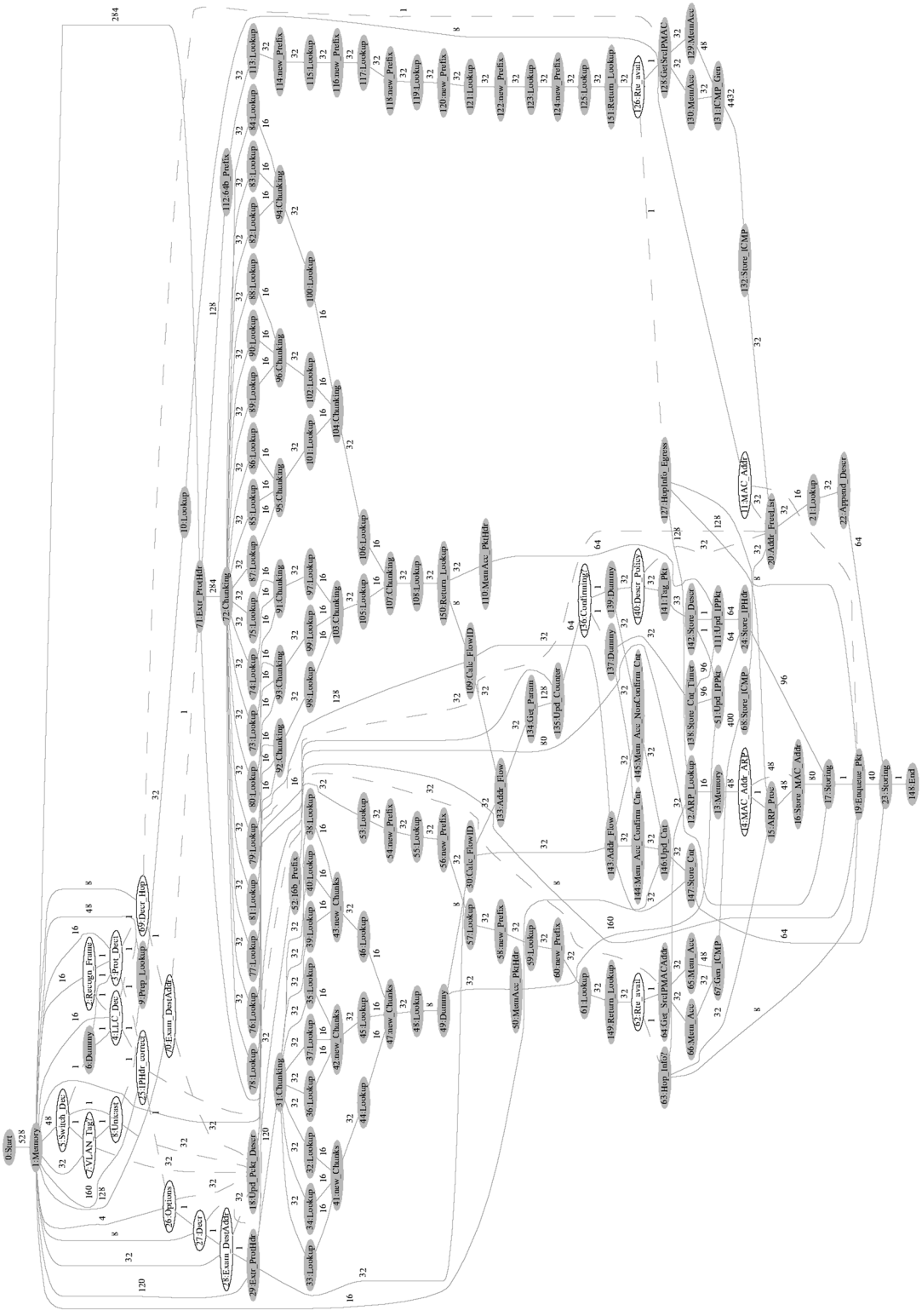


Figure B.1: CPG of *DiffServ*

Number	Description	Processor 1	Processor 2	Shared Memory	Lookup-Engine	Lookup-Engine Memory	Classifier	Classifier Memory	Next-hop Lookup	Next-hop Lookup Memory	Resource Manager	Resource Manager Memory
54	New Prefix	6	6						1			
55	Lookup			2						1		
56	New Prefix	6	6						1			
57	Lookup			2						1		
58	New Prefix	6	6						1			
59	Lookup			2						1		
60	New Prefix	6	6						1			
61	Lookup			2						1		
62	Route available?	4	4									
63	Hop-Info Evaluation	10	10									
64	Fetch Src-IP/MAP Address	2	2									
65	Memory Access			3								
66	Memory Access			2								
67	Generate ICMP Message incl. Checksum	30	45									
68	Store ICMP Message			6								
69	Decrement HOP Count	3	4									
70	Examine Dest-Address	16	16									
71	Extract Protocol Header	6	6									
72	Create Chunks	36	48				1					
73	Lookup			2				1				
74	Lookup			2				1				
75	Lookup			2				1				
76	Lookup			2				1				
77	Lookup			2				1				
78	Lookup			2				1				
79	Lookup			2				1				
80	Lookup			2				1				
81	Lookup			2				1				
82	Lookup			2				1				
83	Lookup			2				1				
84	Lookup			2				1				
85	Lookup			2				1				
86	Lookup			2				1				
87	Lookup			2				1				
88	Lookup			2				1				
89	Lookup			2				1				
90	Lookup			2				1				
91	Create New Chunks	9	12				1					
92	Create New Chunks	9	12				1					
93	Create New Chunks	9	12				1					
94	Create New Chunks	9	12				1					
95	Create New Chunks	9	12				1					
96	Create New Chunks	9	12				1					
97	Lookup			2				1				
98	Lookup			2				1				
99	Lookup			2				1				
100	Lookup			2				1				
101	Lookup			2				1				
102	Lookup			2				1				
103	Create New Chunks	9	12				1					
104	Create New Chunks	9	12				1					
105	Lookup			2				1				
106	Lookup			2				1				
107	Create New Chunks	9	12				1					
108	Lookup			2				1				
109	Flow ID Calculation	30	40									
110	Header Update IP Packet			4								
111	Priority/Flowlabel, Hopcount	20	25									

Table B.9: WCET Table of the Resources for *DiffServ*

Number	Description	Processor 1	Processor 2	Shared Memory	Lookup-Engine	Lookup-Engine Memory	Classifier	Classifier Memory	Next-hop Lookup	Next-hop Lookup Memory	Resource Manager	Resource Manager Memory
112	Creation of 64 bit Prefix	6	6		1							
113	Lookup			2		1						
114	New Prefix	6	6		1							
115	Lookup			2		1						
116	New Prefix	6	6		1							
117	Lookup			2		1						
118	New Prefix	6	6		1							
119	Lookup			2		1						
120	New Prefix	6	6		1							
121	Lookup			2		1						
122	New Prefix	6	6		1							
123	Lookup			2		1						
124	New Prefix	6	6		1							
125	Lookup			2		1						
126	Route available?	4	4									
127	Evaluation of Hop-Info for Egress Port	10	10									
128	Fetch Src-IP/MAP Address	2	2									
129	Memory Access			3								
130	Memory Access			8								
131	Generation of ICMP Message incl. Checksum	48	60									
132	Storing of ICMP Message			10								
133	Create Address for Flow	2	4								1	
134	Fetch Counter			4								1
135	Update Counter	10	15								1	
136	Conforming?	3	3								1	
137	Dummy	1	1								1	
138	Storing of Counter and Compliance Time	2	2								1	
139	Dummy	1	1								1	
140	Policy Differentiation	2	2								1	
141	Tag Packet	3	3								1	
142	Storing Descriptor	2	2								1	
143	Create Address for Flow	2	3								1	
144	Fetch Counter			2								1
145	Fetch Conforming Counter			2								1
146	Update Counter	2	2								1	
147	Storing Counter	2	2								1	
148	End Node	1										
149	Dummy Node	1	1						1			
150	Dummy Node	1	1				1					
151	Dummy Node	1	1		1							

Table B.9: WCET Table of the Resources for *DiffServ*

Results

The results of the *DiffServ* design model for the various scnearios are given. The overall latency is given in time units (T).

DiffServ_Proc			**Overall Latency**
ReCA	without *ESU*	*Cluster_CIN*	1433
		Cluster_Sum	1466
	with *ESU*	*Cluster_CIN*	1466
		Cluster_Sum	1491
ECA_LA1	without *ESU*	*Cluster_CIN*	1496
		Cluster_Sum	1487
	with *ESU*	*Cluster_CIN*	1494
		Cluster_Sum	1471
ECA_LA2	without *ESU*	*Cluster_CIN*	1475
		Cluster_Sum	1483
	with *ESU*	*Cluster_CIN*	1468
		Cluster_Sum	1451

Table B.10: Schedule Latencies for the Scenario *DiffServ_Proc*

DiffServ_1Acc			**Overall Latency**
ReCA	without *ESU*	*Cluster_CIN*	1482
		Cluster_Sum	1499
	with *ESU*	*Cluster_CIN*	1492
		Cluster_Sum	1494
ECA_LA1	without *ESU*	*Cluster_CIN*	1452
		Cluster_Sum	1467
	with *ESU*	*Cluster_CIN*	1511
		Cluster_Sum	1492
ECA_LA2	without *ESU*	*Cluster_CIN*	1486
		Cluster_Sum	1487
	with *ESU*	*Cluster_CIN*	1517
		Cluster_Sum	1534

Table B.11: Schedule Latencies for the Scenario *DiffServ_1Acc*

DiffServ_1AccMem			**Overall Latency**
ReCA	without *ESU*	*Cluster_CIN*	1482
		Cluster_Sum	1401
	with *ESU*	*Cluster_CIN*	1425
		Cluster_Sum	1279
ECA_LA1	without *ESU*	*Cluster_CIN*	1450
		Cluster_Sum	1418
	with *ESU*	*Cluster_CIN*	1436
		Cluster_Sum	1285
ECA_LA2	without *ESU*	*Cluster_CIN*	1483
		Cluster_Sum	1431
	with *ESU*	*Cluster_CIN*	1410
		Cluster_Sum	1285

Table B.12: Schedule Latencies for the Scenario *DiffServ_1AccMem*

DiffServ_4Acc			**Overall Latency**
ReCA	without *ESU*	*Cluster_CIN*	1502
		Cluster_Sum	1548
	with *ESU*	*Cluster_CIN*	1530
		Cluster_Sum	1538
ECA_LA1	without *ESU*	*Cluster_CIN*	1493
		Cluster_Sum	1528
	with *ESU*	*Cluster_CIN*	1545
		Cluster_Sum	1546
ECA_LA2	without *ESU*	*Cluster_CIN*	1491
		Cluster_Sum	1562
	with *ESU*	*Cluster_CIN*	1527
		Cluster_Sum	1554

Table B.13: Schedule Latencies for the Scenario *DiffServ_4Acc*

DiffServ_4AccMem			**Overall Latency**
ReCA	without *ESU*	*Cluster_CIN*	1441
		Cluster_Sum	1240
	with *ESU*	*Cluster_CIN*	1365
		Cluster_Sum	1235
ECA_LA1	without *ESU*	*Cluster_CIN*	1465
		Cluster_Sum	1240
	with *ESU*	*Cluster_CIN*	1384
		Cluster_Sum	1223
ECA_LA2	without *ESU*	*Cluster_CIN*	1486
		Cluster_Sum	1303
	with *ESU*	*Cluster_CIN*	1358
		Cluster_Sum	1249

Table B.14: Schedule Latencies for the Scenario *DiffServ_4AccMem*

Abbreviations and Acronyms

ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
CIN	Common Implementation Nodes
CP	Critical Path
CPG	Conditional Process Graph
ECA	Enhanced Constructive Algorithm
ECP	Enhanced Critical Path
FAST	Fast Assignment using Search Technique
HDL	Hardware Description Language
HW	Hardware
IC	Integrated Circuits
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IP	Intellectual Property
LA	Look-Ahead
LLC	Logical Link Control
MAC	Medium Access Control
ME	Mutual Exclusiveness
NOC	Network on Chip
OAM	Operation and Management
PDU	Protocol Data Unit
ReCA	Reference Constructive Algorithm
RTL	Register-Transfer Level
SLD	System Level Design
SoC	Systems on a Chip
Src-IP	Source IP (Addresse)
SW	Software
T	Time Unit
TGFF	Task Graph For Free
TLM	Transaction Level Modeling

TS	Tabu Search
TTL	Time-to-Live
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLAN	Virtual Local Area Network (IEEE 802.1Q)
WCET	Worst Case Execution Time

Bibliography

[1] P. Arató, S. Juhász, Z. Mann, A. Orbán, D. Papp, "Hardware-Software Partitioning in Embedded System Design," *Proc. of the IEEE International Symposium on Intelligent Signal Processing (ISPACS'03)*, pp. 197-202, September 2003.

[2] J. Aweya, "On the Design of IP Routers Part1: Router Architectures", *Journal of Systems Architecture*, Vol. 46, No. 6, S. 483-511, 2000

[3] J. Becker, "Hardware/Software Codesign", Lecture Notes, *Institut für Technik der Informationsverarbeitung, University of Karlsruhe*, Germany.

[4] V. Berman, "Raising the Level of Abstraction for Design and Verification: SystemC and System Verilog in a Multilanguage Environment", *DesignCon 2005*, February 2005.

[5] T. Blicke, J. Teich, L. Thiele, "System-Level Synthesis Using Evolutionary Algorithms," *Design Automation for Embedded Systems Journal, Kluwer Academic Publishers*, vol. 3, no. 1, pp. 23-58, January 1998.

[6] S. Blake, et al., "An Architecture for Differentiated Services", *RFC2475*, Dezember 1998

[7] W. Brunnbauer, T. Wild, J. Foag, N. Pazos, "A Constructive Algorithm with Look-Ahead for Mapping and Scheduling of Task Graphs with Conditional Edges", *EuroMicro Symposium on Digital System Design 2003*, pp. 98-103, September 2003.

[8] W. Brunnbauer, T. Wild, A. Krug, "Consideration of IP-Modules during Mapping and Scheduling of Task Graphs", *Austrochip 2003*, October 2003.

[9] Cadence Design Systems, Inc., http://www.cadence.com/

[10] Cadence Design's Platform Application Note, *Cadence*, February 2003

[11] Cadence Design's Cierto Virtual Component Co-design (VCC)

[12] L.P. Carloni, F. De Bernardinis, A.L. Sangiovanni-Vincentelli, M. Sgroi, "The Art and Science of Integrated Systems Design", *Proc. of the 32nd European Solidstate Device Research Conference (ESSDERC'02)*, pp. 19-30, September 2002.

[13] W.O. Cesário, D. Lyonnard, G. Nicolescu, Y. Paviot, Y. Sungjoo, A.A. Jerraya, L. Gauthier, M. Diaz-Nava, "Multiprocessor SoC platforms: a component-based design approach," *IEEE Design and Test of Computers*, vol. 19, no. 6, pp. 52-63, November 2002

[14] W.L. Chapman, J. Rozenblit, and A.T. Bahill, "System design is an NP-complete problem," *The Journal of Systems Engineering, INCOSE*, vol. 4, no. 3, pp. 222-229, 2001.

[15] S. Chakraborty, T. Erlebach, S. Künzli, L. Thiele, "Schedulability of event-driven code blocks in real-time embedded systems," *Proc. of the 39th Design Automation Conference (DAC'02)*, pp. 616-621, June 2002.

[16] S. Chakraborty, S. Künzli, L. Thiele, "A General Framework for Analysing System Properties in Platform-Based Embedded System Designs," *Proc. on Design, Automation and Test in Europe (DATE'03)*, pp. 10190-10195, March 2003.

[17] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, P. Sagmeister, "Performance evaluation of network processor architectures: combining simulation with analytical estimation," *Computer Networks*, vol. 41, no. 5, pp. 641-665, April 2003.

[18] B. Cirou, E. Jeannot, "Triplet: A Clustering Scheduling Algorithm for Heterogeneous Systems," ICPP Workshops, pp. 231-236, 2001.

[19] A. Colin, S.M. Petters, "Experimental Evaluation of Code Properties for WCET Analysis," *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS'03)*, pp. 190-199, December 2003.

[20] J.G. D'Ambrosio, X. Hu, "Configuration-level hardware/software partition for real-time embedded systems," *Proc. of the 3th Int. Conference on Hardware Software Codesign (CODES/CASHE'94)*, pp. 34-41, September 1994.

[21] J.A. Darringer, R.A. Bergamaschi, S. Bhattacharya, D. Brand, A. Herkersdorf, J.K. Morrell, I.I. Nair, P. Sagmeister, and Y. Shin, "Early analysis tools for system-on-a-chip design," *IBM Journal of Research and Development*, vol. 46, no. 6, pp. 691-707, November 2002.

[22] B.P. Dave, G. Lakshminarayana, N.K. Jha, "COSYN: Hardware-Software Co-Synthesis of Embedded Systems," *Proc. of the 34th Design Automation Conference (DAC'97)*, pp. 703-708, June 1997.

[23] B.P. Dave, N.K. Jha, "COHRA: hardware-software cosynthesis of hierarchical heterogeneous distributed embedded systems," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, pp. 900-919, October 1998.

[24] R.P. Dick, D.L. Rhodes, and W. Wolf, "TGFF: Task Graphs for Free," *Proc. of the 5th Int. Conference on Hardware Software Codesign (CODES'98)*, pp. 97-101, March 1998, Available at http://www.princeton.edu/~cad/projects.html.

[25] R.P. Dick, N.K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Hierarchical Heterogeneous Distributed Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 920-935, October 1998.

[26] R.P. Dick and N.K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems," *Proc. of the Int. Conference on Computer-Aided Design (ICCAD'01)*, pp. 62-68, November 1998.

[27] R.P. Dick and N.K. Jha, "MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis," *Proc. of Design, Automation and Test in Europe (DATE'99)*, pp. 263-270, March 1999.

[28] R.P. Dick and N.K. Jha, "COWLS: Hardware-Software Co-Synthesis of Distributed Wireless Low-Power Client-Server Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 2-16, January 2004.

[29] A. Doboli, P. Eles, "Scheduling under Data and Control Dependencies for Heterogeneous Architectures," *Proc. of the Int. Conference on Computer Design (ICCD'98)*, pp. 602-608, October 1998.

[30] R. Dutta, J. Roy, R. Vemuri, "Distributed Design-Space Exploration for High-Level Synthesis Systems," *Proc. of the 29th Design Automation Conference (DAC'92)*, pp. 644-650, September 1992.

[31] P. Eles, Z. Peng, K. Kuchcinski, A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search", *Design Automation for Embedded Systems Journal, Kluwer Academic Publishers*, vol. 2, no.1, pp. 5-32, January 1997.

[32] P. Eles, K. Kuchcinski, Z. Peng, P. Pop, A. Doboli, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems," *Proc. on Design, Automation and Test in Europe (DATE'98)*, pp. 132-138, February 1998.

[33] P. Eles, A. Doboli, P. Pop, Z. Peng, "Scheduling with Bus Access Optimization for Distributed Embedded Systems," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 5, pp. 472-491, October 2000.

[34] D.D. Gajski, R. Kuhn, "New VLSI Tools - Guest Editors' Introduction," *IEEE Computer*, vol. 16, no. 12, pp. 11-14, December 1983.

[35] D.D. Gajski, F. Vahid, S. Narayan, J. Gong, "Specification and Design of Embedded Systems," *P T R Prentice Hall*, Englewood Cliffs, NJ, USA, 1994.

[36] D.D. Gajski, "System-Level Design Methodology of SoC Designs", *Slides of Tutorial 3, 9th Asia and South Pacific Design Automation Conference (ASPDAC'04)*, January 2004.

[37] S.H. Gerez, "Algorithms for VLSI Design Automation," *John Wiley & Sons Press*, West Sussex, 1998.

[38] T. Givargis, F. Vahid, J. Henkel, "System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip", *Proc. of the Int. Conference on Computer-Aided Design (ICCAD'01)*, pp. 25-30, November 2001.

[39] F. Glover, "Tabu Search – Part I," *Operations Research Society of America (ORSA) Journal on Computing*, vol. 1, no. 3, pp. 190-206., August 1989.

[40] F. Glover, "Tabu Search – Part II," *Operations Research Society of America (ORSA) Journal on Computing*, vol. 2, no. 1, pp. 4-32, February 1990.

[41] G. Gogniat, M. Auguin, L. Bianco, A. Pegatoquet, "Communication synthesis and HW/SW integration for embedded system design," *Proc. of the 6th International Workshop on Hardware/Software Codesign (CODES'98)*, pp. 49-53, March 1998.

[42] P. Gupta, N. McKeown, "Packet Classification on Multiple Fields", Proc. ACM SIGCOMM 1999, S. 147-160, September 1999.

[43] P. Gupta, "Algorithms for Routing Lookups and Packet Classification", *Dissertation*, Stanford University, December 2000.

[44] P. Gupta, N. McKeown "Algorithms for Packet Classification", *IEEE Network*, Vol. 15, No. 2, S. 24-32, March 2001.

[45] C. Haubelt, J. Teich, K. Richter, R. Ernst, "System Design for Flexibility," *Proc. on Design, Automation and Test in Europe (DATE'02)*, pp. 854-861, March 2002.

[46] J. Hu, R. Marculescu, "Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints," *Proc. on Design, Automation and Test in Europe (DATE'04)*, pp. 234-239, Febuary 2004.

[47] IEEE Standard Verilog Hardware Description Language, *IEEE Standard P1364-2005*, http://www.verilog.com/IEEEVerilog.html

[48] *IETF (The Internet Engineering Task Force) Working Groups*, Differentiated Services (diffserv), http://www.ietf.org/html.charters/diffserv-charter.html

[49] *IETF (The Internet Engineering Task Force) Working Groups* , Integrated Services (intserv), http://www.ietf.org/html.charters/intserv-charter.html

[50] ITRS International Technology Roadmap for Semiconductors 2001 Edition.

[51] ITRS International Technology Roadmap for Semiconductors 2003 Edition.

[52] R. James, B. Storer, "Methods for Solving Subset Sum Problems," *Presentation Slides*, Department of Management, University of Canterbury, Christchurch, New Zealand, http://www.mang.canterbury.ac.nz.

[53] A. Jantsch, "Models of Embedded Computation," Invited contribution in *Embedded Systems, CRC Press*, to appear in 2005.

[54] B. Jeong, S. Yoo, S. Lee, K. Choi, "Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs," *Proc. of the 5th Asia and South Pacific Design Automation Conference (ASPDAC'00)*, pp. 169-174, January 2000.

[55] S.C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, pp. 241-254, September 1967.

[56] H.-P. Juan, D. Gajski, V. Chaiyakul, "Condition Graphs for HighQuality Behavioral Synthesis," *Technical Report #94-32, Department of Information and Computer Science, University of California, Irvine*, August 1994.

[57] A. Kalavade, E.A. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," *Proc. of the 1th Int.*

Conference on Hardware Software Codesign (CODES'94), pp. 42-48, September 1994.

[58] A. Kalavade, E.A. Lee, "The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling, and Implementation-bin Selection," *Design Automation for Embedded Systems Journal, Kluwer Academic Publishers*, vol. 2, no.2, pp. 125-163, January 1997.

[59] K. Keutzer, S. Malik, R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design", *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 19, no. 12, December 2000, pp.1523-1543.

[60] Andreas Krug, "Unterstützung von IP-Modulen in Verfahren für die Abbildung und das Scheduling von Prozessgraphen", *Diplomarbeit (Master's Thesis)*, September 2003.

[61] Y.-K. Kwok, I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.

[62] Y.-K. Kwok, I. Ahmad, J. Gu, "FAST: A Low Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," *Proc. of the Int. Conference on Parallel Processing (ICPP'96)*, vol. II, pp. 150-157, August 1996.

[63] Y.-K. Kwok, I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using A Parallel Genetic Algorithm," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 58-77, November 1997.

[64] Y.-K. Kwok, I. Ahmad, "FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 147-159, February 1999.

[65] Y.-K. Kwok, I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406-471, December 1999.

[66] Y.-K. Kwok, I. Ahmad, "Link Contention-Constrained Scheduling and Mapping of Tasks and Messages to a Network of Heterogeneous Processors," *Cluster Computing*, vol. 3, no. 2, pp. 113-124, 2000.

[67] E.D. Lagnese, D.E. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 7, pp. 847-860, July 1991.

[68] K. Lahiri, A. Raghunathan, S. Dey, "Fast performance analysis of bus-based system-on-chip communication architectures," *Proc. of the Int. Conference on Computer-Aided Design (ICCAD'99)*, pp. 566-573, November 1999.

[69] K. Lahiri, A. Raghunathan, G. Lakshminarayana, S. Dey, "Communication architecture tuners: a methodology for the design of high-performance

communication architectures for systems-on-chips," *Proc. of the 37th Design Automation Conference (DAC'00)*, pp. 513-518, June 2000.

[70] K. Lahiri, A. Raghunathan, S. Dey, "Efficient Exploration of the SoC Communication Architecture Design Space," *Proc. of the Int. Conference on Computer-Aided Design (ICCAD'00)*, pp. 424-430, November 2000.

[71] K. Lahiri, A. Raghunathan, S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 768-783, June 2001.

[72] Y.-T.S. Li, S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," *Proc. of the 32th Design Automation Conference (DAC'95)*, pp. 456-461, June 1995.

[73] A. Maxiaguine, S. Künzli, L. Thiele, "Workload Characterization Model for Tasks with Variable Execution Demand," *Proc. on Design, Automation and Test in Europe (DATE'04)*, pp. 1040-1045, February 2004.

[74] M.C. McFarland, T.J. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 9, pp. 938-950, September 1990.

[75] D. Mohanty, R. Mahapatra, G. Choi, "A Design Space Exploration Framework in Multiprocessor SoC Codesign", *Proc. of the Workshop on RTSS Embedded Systems*, December 2001.

[76] K. Nichols et al., "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", *RFC2474*, Dezember 1998

[77] P.G. Paulin, J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 6, June 1989.

[78] S.M. Petters, G. Färber, "Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible," *Proc. of the 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA'99)*, pp. 442-449, Dezember 1999.

[79] S.M. Petters, "Comparison of Trace Generation Methods for Measurement Based WCET Analysis," *Proc. of the 3rd Int. Workshop on Worst-Case Execution Time Analysis, (WCET'03)*, pp. 61-74, July 2003.

[80] QNX Software Systems, *http://www.qnx.com/*.

[81] A. Sangiovanni-Vincentelli, G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems", *IEEE Design & Test of Computers*, vol. 18, no. 6, pp.23-33, November 2001.

[82] M. Schmitz, B. Al-Hashimi, P. Eles, "Energy-Efficient Mapping and Scheduling for DVS Enabled Distributed Embedded Systems," *Proc. on Design, Automation and Test in Europe (DATE'02)*, pp. 514-521, March 2004.

[83] M. Schwiegershausen, P. Pirsch, "A system level design methodology for the optimization of heterogeneous multiprocessors," *Proc. of the 8th International Symposium on System Synthesis (ISSS'95)*, pp. 162-169, September 1995.

[84] D. Shin, J. Kim, "Power-aware scheduling of conditional task graphs in real-time multiprocessor systems," *Proc. of the Int. Symposium on Low Power Electronics and Design (ISLPED'03)*, pp. 408-413, August 2003.

[85] G.C. Sih, E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175-187, February 1993.

[86] M.J.S. Smith, "Application-Specific Integrated Circuits," *Addison Wesley Professional*, 1997.

[87] Synopsys, Inc., *http://www.synopsys.com*

[88] SystemC Community, *http://www.systemc.org/*

[89] J. Teich, "Digitale Hardware/Software-Systeme- Synthese und Optimierung," *Springer-Verlag Berlin Heidelberg*, 1997.

[90] J. Teich, "Embedded System Synthesis and Optimization," *Invited paper for the Proceedings of the Workshop on System Design Automation (SDA'00)*, VDE-Verlag, pp. 9-22, March 2000.

[91] L. Thiele, S. Chakraborty, M. Gries, S. Knzli, "Design Space Exploration of Network Processor Architectures," *First Workshop on Network Processors at the 8th International Symposium on High Performance Computer Architecture (HPCA8*), February 2002.

[92] L. Thiele, S. Chakraborty, M. Gries, S. Künzli, "A framework for evaluating design tradeoffs in packet processing architectures," *Proc. of the 39th Design Automation Conference (DAC'02)*, pp. 880-885, June 2002.

[93] H. Topcuoglu, S. Hariri, M.-Y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260-274, March 2002.

[94] K. Vallerio, N.K. Jha, "Task graph transformation to aid system synthesis," *Proc. of the Int. Symposium on Circuits and Systems*, pp. 695-698, May 2002.

[95] IEEE Standard VHDL Language Reference Manual, *IEEE-SA Standards Board*, January 2000.

[96] VSI Alliance System-Level Design Development Working Group, VSI Alliance Model Taxonomy Version 2.1 (SLD 2 2.1), July 2001.

[97] K. Wakabayashi, H. Tanaka, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors," *Proc. of the 29th Design Automation Conference (DAC'92)*, pp. 112-115, June 1992.

[98] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, "Scalable High Speed IP Routing Lookups", *Proc. of ACM SIGCOMM 1997*, S. 25-37, 1997.

[99] T. Wild, W. Brunnbauer, J. Foag, N. Pazos, "Integrating On-Chip Communication in HW/SW-Partitioning of Networking Systems-on-Chip", *Proc. Int. Workshop on IP-Based SoC Design*, December 2001.

[100] T. Wild, W. Brunnbauer, J. Foag, N. Pazos, "Mapping and Scheduling for Architecture Exploration of Networking SoCs", *Proc. 16th Int. Conference on VLSI Design*, January 2003.

[101] T. Wild, "Ein rekursives Verfahren zur Abbildung und zum Scheduling von Prozess-Graphen mit Kontrollabhängigkeiten", *PhD Thesis, Lehrsuhl für Integrierte Schaltungen, Techische Universität München*, July 2003.

[102] A. Winckler, "A Distributed Look-Ahead Algorithm for Scheduling Interdependent Tasks," *Proc. of the Int. Symposium on Autonomous Decentralized Systems (ISADS'93)*, pp. 190-197, March 1993.

[103] Wind River, Inc., http://www.windriver.com/

[104] T. Wolf, M.A. Franklin, "CommBench - a telecommunications benchmark for network processors," *Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, pp. 154–162, April 2000.

[105] D. Wu, B. Al-Hashimi, P. Eles, "Scheduling and Mapping of Conditional Task Graph for the Synthesis of Low Power Embedded Systems," *IEE Proceedings on Computers and Digital Techniques*, vol. 150, no. 5, pp. 303-312, September 2003.

[106] M.-Y. Wu, W. Shu, J. Gu, "Efficient Local Search for DAG Scheduling," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 617-627, 2001.

[107] Y. Xie, W. Wolf, "Allocation and scheduling of conditional task graph in hardware/software co-synthesis", *Proc. on Design, Automation and Test in Europe (DATE'01)*, pp. 620-625, March 2001.

[108] Y. Zhang, X. Hu, D. Z. Chen, "Task scheduling and voltage selection for energy minimization," *Proc. of the 39th Design Automation Conference (DAC'02)*, pp. 183-188, September 2002.

[109] E. Zitzler, L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257-271, November 1999.

[110] V.D. Zivkovic, P. Lieverse, "An Overview of Methodology and Tools in the Field of System-Level Design", *in F. Deprettere, J. Teich, S. Vassiliadis (Eds.), "Embedded Processor Design Challenges, Systems, Architectures, Modelling, and Simulation - SAMOS"*, pp. 74-88, September 2002.