**The Thesis Committee for Ashmita Sinha**
**Certifies that this is the approved version of the following thesis:**

# Multi-Objective Trade-Off Exploration For Cyclo-Static And Synchronous Dataflow Graphs

**APPROVED BY**
**SUPERVISING COMMITTEE:**

**Supervisor:**

Andreas Gerstlauer, Supervisor

Lizy Kurian John

# Multi-Objective Trade-Off Exploration For Cyclo-Static And Synchronous Dataflow Graphs

**by**

**Ashmita Sinha, B.Tech.**

**Thesis**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

**The University of Texas at Austin**

**August 2012**

Dedicated to my parents

## Acknowledgements

I would like to take this opportunity to express my sincere gratitude to my supervisor, Dr. Andreas Gerstlauer, for all his support and guidance throughout this thesis work. This research work would not have been the same without his advice and directions. It was through his encouragement during EE382V that I found what I wanted to do for my thesis. He has helped me in developing a strong background in this field.

I would like to thank Prof Lizy John for providing her suggestions in this work. I would also like to extend my gratification to Jacob Kornerup and Mike Trimborn from National Instruments, Austin, for their active involvement in this project. Their feedback during the bi weekly meetings has been invaluable.

Last but not the least, I would also like to thank Jing Lin who helped me in understanding the concepts during the initial phase of my work. I had a great time working with Aman Arora, who joined this project in its early stage and volunteered his time and energy. I appreciate his contribution that helped me in getting started with my research work.

# Abstract


## Multi-Objective Trade-Off Exploration For Cyclo-Static And Synchronous Dataflow Graphs

Ashmita Sinha, M.S.E.

The University of Texas at Austin, 2012

Supervisor: Andreas Gerstlauer

Many digital signal processing and real-time streaming systems are modeled using dataflow graphs, such as Synchronous Dataflow (SDF) and Cyclo-static Dataflow (CSDF) graphs that allow static analysis and optimization techniques. However, mapping of such descriptions into tightly constrained real-time implementations requires optimization of resource sharing, buffering and scheduling across a multi-dimensional latency-throughput-area objective space. This requires techniques that can find the Pareto-optimal set of implementations for the designer to choose from. In this work, we address the problem of multi-objective mapping and scheduling of SDF and CSDF graphs onto heterogeneous multi-processor platforms. Building on previous work, this thesis extends existing two-stage hybrid heuristics that combine an evolutionary algorithm with an integer linear programming (ILP) model to jointly optimize throughput, area and latency for SDF graphs. The primary contributions of this work include: (1) extension of the ILP model to support CSDFGs with additional buffer size optimizations; (2) a further optimization in the

ILP-based scheduling model to achieve a runtime speedup of almost a factor of 10 compared to the existing SDFG formulation; (3) a list scheduling heuristic that replaces the ILP model in the hybrid heuristic to generate Pareto-optimal solutions at significantly decreased runtime while maintaining near-optimality of the solutions within an acceptable gap of 10% when compared to its ILP counterparts. The list scheduling heuristic presented in this work is based on existing modulo scheduling approaches for software pipelining in the compiler domain, but has been extended by introducing a new concept of mobility-based rescheduling before resorting to backtracking. It has been proved in this work that if mobility-based rescheduling is performed, the number of required backtrackings and hence overall complexity and runtime is less.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1 Introduction

The emergence of multimedia and wireless applications as growth leaders has created an increased demand for embedded systems with high performance. This has led to a shift in the industry towards multi-processor system-on-chip (MPSoC) designs consisting of heterogeneous multi-processor and multi-core architectures. In many embedded application domains, an essential component is their real-time streaming behavior, e.g. for critical signal processing in media, analytics or communications. Next to functional correctness, these applications have stringent non-functional requirements, such as throughput, latency, or cost. Capturing these design concerns globally while making optimal design decisions at an early system-level stage is therefore a big challenge.

## 1.1. Dataflow Modeling

Digital signal processing applications are non-terminating in nature, continuously processing infinite streams of input and output data. Dataflow models of computation (MoCs) are widely used for modeling and analyzing such streaming applications on multiprocessor platforms. Dataflow applications are composed of several independent tasks that can run in an autonomous way as soon as they have the appropriate data available at all their inputs. Synchronous dataflow graphs (SDFGs) are used to model systems that have a fixed rate of data production and consumption. This makes static analysis a possibility. Cyclo static dataflow graphs (CSDFGs) are an extension of SDFGs to model a wider range of signal processing applications whose rate of production and consumption of data streams changes cyclically. The tasks or computation blocks of streaming applications are modeled as actors in the directed dataflow graph, while the communication channels allowing the transfer of data (tokens) are represented as unbounded FIFO edges between actors.

## 1.2. Multi-Processor Mapping

Mapping and scheduling of actors onto processing elements (PEs) of a multi-processor target architecture form one of the most critical tasks in the design process. Often, a platform is pre-designed and only supports limited configurability. Hence, optimization options are explored by modifying the mapping instead. Mapping a dataflow graph typically involves allocation of a set of PEs and binding the actors onto them. However, evaluating the optimality of a mapping solution amongst various possible options requires extensive exploration. As such, the most practical approach to solving the mapping problem is to develop heuristics for approximate mapping algorithms [9].

Scheduling involves determining what would be the best start time for each actor on its PE either after or while the mapping is decided. Many constraint programming and other scheduling heuristic have been proposed. Most of these try to optimize either throughput or buffer size requirements of the design in their solutions. There is always a tradeoff between throughput, buffer size requirements, total area and latency across a large design space of implementation options. However to achieve high performance, a streaming application should have a highly optimized execution path that minimizes latency in addition to optimizing throughput and buffer in a balanced way [11]. Inclusion of latency into the optimization gives hard real-time guarantees on the task completion deadline. Optimization across a multi-dimensional design and objective space and generating a complete set of Pareto-optimal solutions is the desired way to help designers in making design decisions. In this thesis work, we address this multi objective optimization problem for streaming applications modeled as SDF and CSDF graphs.

In Jin et al. [11], a hybrid evolutionary algorithm (EA) and integer linear programming (ILP) model has been presented to deal with this multi-objective optimization problem for SDFGs. In this thesis, this technique has been extended to include a wider range of applications modeled as CSDFGs. Furthermore, our scheduling ILP optimizes buffer requirements of the graph in addition to just

minimizing latency as was done in the earlier work. In addition, methods to further optimize the ILP model have been explored and implemented so as to obtain lower run times while still guaranteeing the same optimal solutions. We develop an improved ILP formulation based on an approach presented in Govindrajan's work [3] that tries to establish a periodic relationship between different iterations of actors statically before beginning to schedule. This allows to get rid of the startup phase that is needed in Jin et al. [11], thereby allowing it to reduce complexity of the ILP formulation and find a periodic schedule faster.

The drawback of ILP-based models in general, however, is that the design complexity increases exponentially with an increase in the graph size. This adversely affects the run times. To tackle this problem, an alternative list scheduling (LS) based algorithm has been developed in this work. It takes advantage of modulo scheduling approaches for software pipelining used in the compiler domain to similarly generate periodic schedules that minimize latency [9]. A hybrid EA-LS based mapping-scheduling model has significantly improved run time efficiency with solutions that are within an acceptable optimality gap when compared to corresponding EA-ILP based approach.

### 1.3. Thesis organization

The rest of the thesis is organized as follows: Chapter 2 presents a literature survey of existing approaches for dealing with the task of multiprocessor design optimizations adopted in different domains. Chapter 3 briefly discusses the dataflow model of computation with an emphasis on SDF and CSDF based models. It also introduces the actual problem definition that has been dealt with as part of this work, with an overview of the two-stage heuristic model used for mapping and scheduling of dataflow applications. Chapter 4 then highlights the modifications that have been made to the existing mapping heuristic described in Jin et al. [11] in order to take into account buffer and area optimizations, while Chapter 5 describes the scheduling techniques that have been developed as part of this work in detail. This includes both

integer linear programming (ILP) and list scheduling (LS) based models. Chapter 6 characterizes the heuristic models presented in this work by applying them to randomly generated graphs and a practical example of an OFDM transmitter and decoder. Finally, Chapter 7 concludes the thesis with a summary and outlook on potential future work.

# Chapter 2 Related Work

The problem of finding an optimal mapping and scheduling is known to be NP-complete and has been extensively looked at by the researchers in the past. Different formulations, e.g. using constraint programming, integer linear programming or heuristics for mapping and scheduling have been in existence in different domains. Problems with smaller size can be efficiently handled by methods that deterministically provide the optimal solution by exhaustively exploring the solution space and returning the theoretical optimum. ILP-based formulations belong into this category. For larger design sizes, however, these exhaustive searches become difficult and at times impossible to apply. Another drawback of ILP-based scheduling is that it cannot detect a combination of infeasible constraints without performing an exhaustive search, thereby resulting in large run times. Heuristics, on the other hand, are pseudo-random search and optimization techniques that perform the exploration and exploitation of solution space based on learned or encoded experience. Therefore, more efficient but non-optimal heuristics are more applicable in such cases where a reasonable quality solution needs to be provided in a relatively shorter time. This literature survey explores different approaches from the high-level synthesis domain, the compiler domain and the system-level domain. These domains mainly differ in their input model, but the basic problem of mapping and scheduling the input model is built upon similar concepts.

In the compiler and high level synthesis (HLS) domains, the design problem is modeled as a directed acyclic graph (DAG), which is equivalent to an acyclic homogeneous synchronous dataflow (HSDF) graph. A HSDFG is a special type of SDFG where all token production and consumption rates are unity. Every SDFG can be converted into an equivalent HSDFG [2], but the size of the resulting graph is much larger than the original one and hence results in an increased complexity. Researchers

have explored several techniques, which range from ILP-based optimal techniques to heuristics for finding periodic schedules under given constraints for such models. In Stoutchinin et al. [4] propose an ILP formulation to model software pipelining for the MIPS R8000 processor. It uses the concept of a circular reservation table for modeling resource sharing. A similar and more widely used approach is that of the so-called modulo reservation table (MRT) for modeling of resource constraints, which was initially introduced by Lam [6] for the compiler domain. The iterative modulo scheduling technique developed by Rao [7] builds upon Lam's model and develops a list scheduling algorithm for generating hardware-constrained periodic schedules. The list scheduling algorithm developed as part of this work uses the concepts introduced in Rao's approach, which is discussed in detail in Chapter 5.

In the high-level synthesis domain, Goossens et al. [8] present a two-stage list scheduling algorithm for a given DAG such that it maximizes the throughput. To ensure periodicity of the schedule on a hardware-constrained model, it uses the concept of loop folding. Kim and Chang [15] present a similar list scheduling algorithm for minimizing latency of a given hardware-constrained graph. Chiu et al. [14] present a novel tile-piecing algorithm for scheduling of a given DAG while attempting to find a periodic schedule for a given period. It also tries to optimize the number of resource used. While these algorithms try a higher period if they fail to find a valid periodic schedule in the first attempt, Rao's algorithm [7] gains an advantage over them owing to its backtracking property. The probability of finding a valid schedule increased with backtracking and hence this work is based upon the framework described in Rao's iterative modulo scheduling algorithm [7].

In the system-level domain, where applications are modeled as variants of synchronous dataflow graphs (SDFGs), similar ILP-based and heuristic approaches have been proposed. Lee et al. [10] propose an assembly line scheduling algorithms from operations research for generating Periodic Admissible Parallel Schedules (PAPS). Geilen et al. [1] present a scheduling heuristic for generating the Pareto-front of throughput versus buffer size for CSDFGs. No hardware constraints are assumed in

this case. For periodicity checks, the architectural state of the graph needs to be saved at each time instance. Also, the problem has to be modeled for sufficient time before it can reach a stable periodic phase. Chen and Doemer use a similar technique to check for the periodic phase [12]. Their algorithm also tries to minimize buffer requirements for a given throughput target and uses a list scheduling framework. Both approaches require a startup phase and associated overhead before the periodic phase can be reached. Also the requirement of saving and comparing the entire architectural state at each time instance as a check for periodicity further increases the algorithm's complexity. Groot et al. [5] try to iteratively find a periodic schedule for a given throughput-constrained dataflow graph taking into account the scheduling range of each actor. Only once the graph has been scheduled, however, a mapping is found to optimize the number of hardware resources. This is, however, not applicable to most practical applications where hardware resources are limited and constrained.

Among optimal and exhaustive approaches, Zhu et al. [12] use constraint programming for mapping throughput-constrained SDFGs onto multi-processor platforms under minimum buffer requirement objectives. Govindrajan et al. [3] present an ILP-based scheduling algorithm for minimizing buffer space requirements under throughput constraints of signal processing applications models as regular stream flow graphs (RSFGs). Their approach is based on multi-rate software pipelining to allow for maximum overlapping of operations from successive iterations, subject only to precedence constraints caused by data dependencies. As such, the approach is a single-objective optimization formulation that does not consider any hardware constraints. The improved ILP presented as part of this thesis extends the algorithm used in Govindrajan's work [3] to SDFGs and uses similar hardware constraint modeling technique as in Rao's algorithm [7].

Most of the hardware-constraint aware algorithms discussed above generate schedules for a single mapping decision of the graph. A significant amount of work has been on generating Pareto-optimal solution sets for design space exploration. genetic algorithms (GAs), inspired by the natural process of evolution, have been

proven to be effective in generating Pareto fronts in multi-objective optimization problems [16]. The model implemented for SDFGs by Jing et al. [11] is based upon a GA-ILP combination for generating the Pareto front. It builds the basis for the work in this thesis and is discussed in detail in Chapter 3. As mentioned earlier, in this work, the model has been extended to support CSDFGs with enhancements such as new ILP and list scheduling formulations and inclusion of buffer minimization in the objective function.

# Chapter 3 Dataflow Synthesis

A dataflow model of computation offers many attractive features for parallel processing. Its model of execution is asynchronous, i.e. execution of an operation is based upon the availability of its inputs, which makes synchronization of parallel activities implicit. Also the model doesn't impose any constraint on sequencing except for the existing data dependencies. It is for these reasons that this model of computation has widely been adopted for expressing real time streaming applications.

## 3.1. Dataflow Models

In a dataflow model processes are broken down into atomic blocks of execution called actors. Actors are connected into a network using unidirectional unbounded FIFOs (first in first out queues) with tokens. Synchronous dataflow (SDF) and cyclo static dataflow (CSDF) models are two variants of this that are widely used for modeling of signal processing applications. In following sections basic concepts of SDF and CSDF will be discussed.

### 3.1.1. Synchronous Dataflow Graphs

A synchronous dataflow graph (SDFG) is a directed graph where *nodes* (referred to as actors) represent the operations that communicate with each other by passing ordered streams of data-elements over their *edges* (referred to as channels). The terms *node* and *actor* are used interchangeably for operations throughout this thesis. An atomic data element used for communication between actors is called *token.* The nodes or actors act upon the input data stream or tokens and transform them to output tokens. In an SDFG, each actor is capable of producing or consuming multiple tokens of data in every firing. The number of tokens produced or consumed by each actor in each firing is predetermined and fixed. This makes it possible to statically

construct a finite schedule that can be repeated periodically to implement a process network that operates on infinite streams of data tokens.

A SDFG can be represented by $G = (V, E, d_u, p_e, c_e)$ such that

- $V$ is the set of operations denoted by nodes in the graph
- $E$ is the set of directed edges that connect nodes in the graph. These edges decide the dependencies between the nodes or the operations
- $d_u$ represents the computation time of actor u
- $p_e$ denotes the number of tokens produced by the source node on edge $e$ every time it is fired
- $c_e$ denotes the number of tokens consumed by the sink node on edge $e$ every time it is fired

A valid schedule for a SDFG can be found only if it is consistent and live. A graph is said to be live if it doesn't have deadlocks. Initial tokens may need to be added to edges to ensure liveness of the graph. A graph is said to be consistent if it has a repetition vector. Repetition vector represents number of firings per actor that brings the graph back to the distribution of tokens before the firings. Also the repetition vector for a consistent graph is the smallest unique non-trivial repetition vector. Once all the actors of the graph have executed their repetition vector times, the graph is said to have completed one *iteration*.

An example of an edge $(u,v)$ is shown below in Figure 3.1, where actor $u$ is the source and actor $v$ is the sink. When actor $u$ fires, it requires $d_u$ time slots to execute and produces $p_{uv}$ tokens on its output edge $(u, v)$ at the end of its execution. Similarly actor $v$ requires $d_v$ time slots to execute and consumes $c_{uv}$ tokens from edge $(u, v)$ at the beginning of its execution. $N_u$ and $N_v$ are the repetition counts of actors' $u$ and $v$ respectively. Repetition counts represent the number of times each actor will execute in a iteration.


Figure 3.1: Example of synchronous dataflow graphs (SDFG).

### 3.1.2. Cyclo Static Dataflow Graphs

The cyclo static dataflow graph (CSDFG) paradigm is an extension of synchronous dataflow graphs. These graphs are more expressive than SDFGs and a larger class of applications can be modeled and more detailed dependencies can be included using them. A CSDFG is more versatile because it supports algorithms with a cyclically changing behavior. Since this behavior is predetermined, it is still possible to schedule CSDFGs statically.

CSDFGs generalize SDFGs by allowing the number of tokens consumed and produced by an actor to vary from one firing to the next in a cyclic pattern. In other words, each actor executes as a sequence of phases, which is repeated in time. Since these patterns are periodic and predictable, it is still possible to construct periodic schedules using techniques based on those developed for SDFGs [19]. An actor may have different production rates, consumption rates and execution times in different phases. Following are the modified notations applicable for CSDFG and different from SDFG:

- $\phi_u$ and $\phi_v$ represent the total number of phases for actors $u$ and $v$ respectively
- $d_{ux}$ represents the computation time of actor u when it is executing in its $x^{th}$ phase, where $x \in [0, \phi_u\text{-}1]$
- $p_{ex}$ denotes the number of tokens produced by source node on edge $e$ in its $x^{th}$ phase, where $x \in [0, \phi_u\text{-}1]$

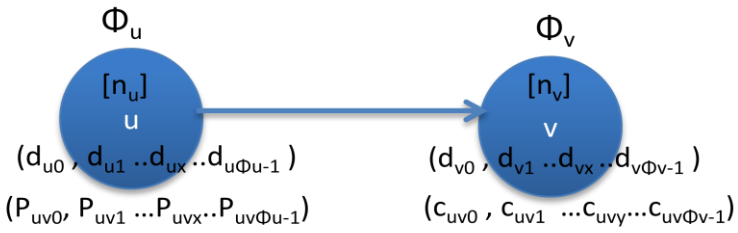


Figure 3.2: Example of cyclo static dataflow graph (CSDFG).

- $c_{ey}$ denotes the number of tokens consumed by sink node on edge $e$ in its $y^{th}$ phase, where x ∈ [0, $\phi_v$-1]

Figure 3.2 below shows an example of an edge *(u, v)* in CSDF graph. Actors *u* and *v* are shown to have total $\phi_u$ and $\phi_v$ phases with production rate, consumption rate and execution times shown in their $x^{th}$ and $y^{th}$ phases respectively. Repetition counts of *u* and *v* are $n_u$ and $n_v$ respectively i.e. in each iteration actors u and v will fire $n_u$ and $n_v$ number of times. Repetition count of an actor is a multiple of its total number of phases. Phases are executed sequentially on each firing of the actor. At any $k^{th}$ firing of actor *u*, its phase will be ($k$ mod $\phi_u$)$^{th}$ phase. In its $x^{th}$ phase, where *x* belongs to range [1, $\phi_u$], actor *u* produces $p_{uvx}$ tokens and requires $d_{ux}$ time cycles to complete its execution. The consumption behavior of actor v is similar. The rest of the definitions of the concepts are the same as that for SDFG.

### 3.2. Problem Formulation

As part of this thesis work, techniques have been explored and developed for multi-objective optimization of the mapping of real time streaming applications, modeled as dataflow graphs, onto heterogeneous multi-processor platforms. This involves allocation of a set of execution resources (i.e., processing elements, PEs), binding of dataflow actors to these PEs, and scheduling of actor execution order among the (shared) resources. Work in this thesis is based upon the basic framework described in Jing et al. [11] for mapping and scheduling of SDF graphs. In this

previous work, the design space exploration algorithm is folded into a two-stage heuristic-ILP combination wherein an evolutionary algorithm (EA) is used to find a mapping while an ILP based scheduling model is used for scheduling the dataflow graph. The result set maps to a Pareto front that charts throughput, latency and cost tradeoffs.

The contribution of this thesis work is extending the approach described by Jing et al. [11] to CSDF graphs, modifying the ILP based scheduling for SDFGs so as to further reduce the run times and finally developing a list scheduling based heuristic for completely replacing the ILP based scheduling model in order to tackle the ILP's exponential increase in design complexity and run time. Fundamentals of EA based heuristic are given in tutorials [11] and [22]. In order to model buffer spaces additional chromosomes have been added in this work. Also to make the mapping model more realistic, IP-type of processing elements are supported such that an actor can be mapped to a particular PE only if their types match or else the EA rejects the mapping.

Figure 3.3 shows the overall EA and ILP/LS based scheduling heuristic that is used for generating a Pareto-optimal front for design space exploration for a given dataflow graph. In each generation, the EA produces a mapping and allocates buffer space for the given input model. Period and area are computed as objective functions in the EA for the corresponding mapping. Period, as the name suggests, represents the period, i.e. the inverse of the throughput of the graph. Area represents the total area required by the processing elements selected in the given mapping and buffer space denotes the total buffer memory allocated for use by the graph. ILP-based or list schedulers take period and buffer space as input constraints and return a schedule that minimizes latency for the given generation of mappings. In the process, the final schedule will also determine the required buffer size, i.e. the actual amount of buffer space occupied by executing the graph in the selected order. These objectives from the EA and scheduler are then evaluated for fitness and used in the evolution process for generating the next generation of chromosomes. Note that in case of ILP based

scheduling models, buffer space can be minimized along with latency as one of the objectives instead of only taking it as a constraint.

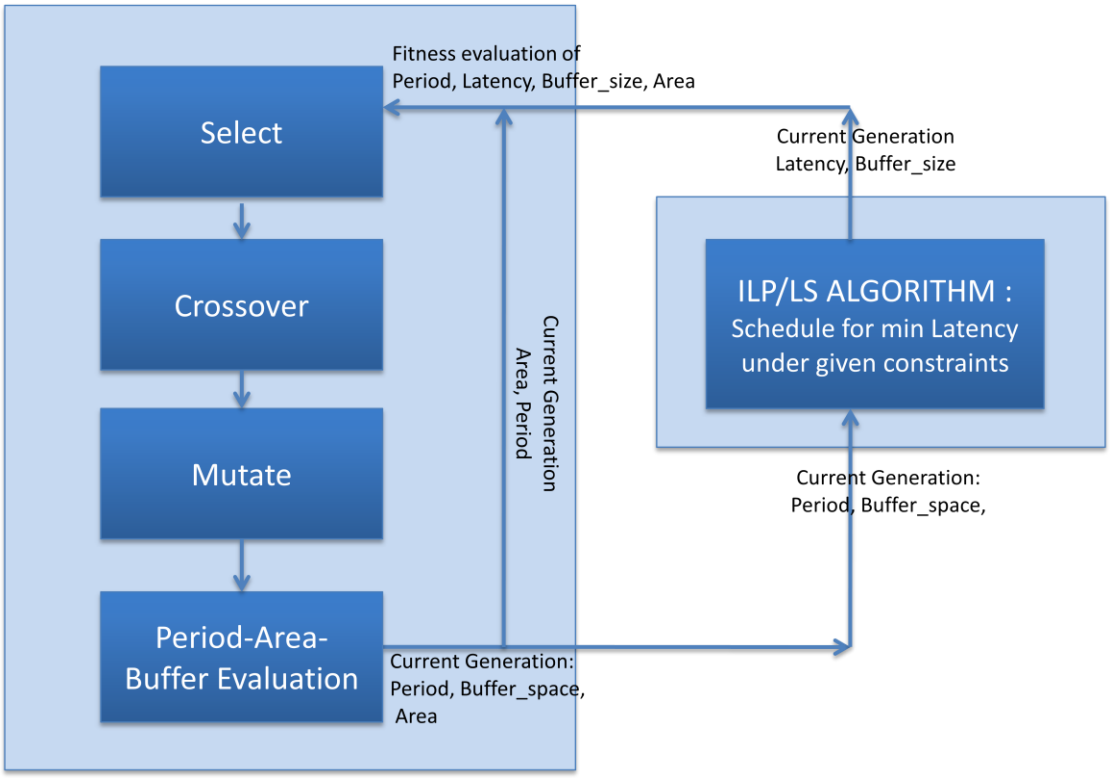


Figure 3.3: Hybrid GA+ILP/LS based dataflow mapping heuristic.

# Chapter 4 Genetic Algorithm for Mapping & Binding

As mentioned earlier, an evolutionary algorithm is used in the two-stage hybrid heuristic for mapping and binding the dataflow graph onto the heterogeneous processor model. A population of chromosomes, which in EA's terminology represents a set of one-to-one mappings to the possible processor partitions, reproduces and evolves over multiple generations under environmental pressure posed by the fitness selection based on achievable throughput, area, latency and buffer sizes [11]. This chapter discusses the input model to the EA along with the specific modifications that have been made to the existing model [11]. The two stage mapping-scheduling model assumes that static analysis has already been performed for the dataflow graph for ensuring its consistency and liveness. Hence, besides other parameters, repetition count of each actor (consistency) and initial tokens on each edge (liveness) are given as inputs to the model.

In a dataflow model of computation, semantics dictate that channels have unbounded storage space, which is not practically applicable. In this thesis work, the existing GA based mapping model has been modified in order to optimize the buffer space in addition to period and area objectives. Bounded storage space for channels can be realized in different ways. One option is to use a memory that is shared between all channels. The required storage space for the execution of a dataflow graph is then determined by the maximum number of simultaneously tokens stored throughout the execution of the graph. Murthy et al. use this assumption to schedule SDFGs with minimal storage space [17]. This is a logical choice for single processor systems in which actors can always share the memory space. A second option is to use a separate memory for each channel, so empty space in one cannot be used for another. This assumption is logical in the context of multiprocessor systems, as memories are not always shared between all processors. The channel capacity must be

determined per channel over the entire schedule, and the total amount of memory required is obtained by summing capacities.

The EA generates an upper bound on the buffer space requirement in each generation along with the mapping. In case of a shared memory, the generated upper bound is for the total buffer space requirement of the graph while for distributed memory it is an upper bound for buffer space on each edge of the graph. With ILP based scheduling models for SDFGs and CSDFGs, a shared memory model is assumed, where the ILP solver takes in the total buffer space as a constraint and further attempts to minimize it along with the latency. Later, with the list scheduling based model, separate memory for each channel is realized where an upper bound of buffer space on each edge is taken as a constraint. Either of the two types of buffer optimizations could be used interchangeably in the ILP and heuristic based schedulers. The only change would be in the possible set of decision variables being used for modeling of buffer space in the EA, as will be described later in Section 4.2.

### 4.1. Target and Mapping Model

For the resource model in this thesis work, an FPGA type architecture is assumed as target onto which heterogeneous multi processors are synthesized. Total number of block RAMs (BRAMs), DSP processors and slices for the FPGA are given that define the upper limit of the area values possible. In this implementation model, IP type of the processing elements have been added, e.g. to allow for pre-designed FIR, FFT etc. PEs. Based upon its specifications, each IP requires a pre-defined and fixed number of BRAMs, slices and DSPs. The EA takes in all these details along with the execution profile, i.e. execution time of each actor on different PEs. Individual PEs can be pipelined in order to further improve the throughput of the overall graph. For this, initiation intervals (*II*s) of each PE are given as input. Also, the following assumptions have been made in this work:

- Unique mapping of actors to PEs is assumed, i.e. an actor cannot be mapped to multiple PEs.

- On each PE actors execute sequentially, i.e. two actors mapped to the same PE will not begin their execution simultaneously.
- In case of CSDF graphs, the repetition vector of each actor is an integer multiple of its total number of phases.
- Inter- and intra-processor communication overhead is assumed to be folded into actor execution times and no separate modeling is done for it. Communication-aware mapping can be independently folded into this work, e.g. using an approach as described in Jing et al. [18], but are out of the scope of this thesis.

A comprehensive list of input variables and parameters needed by this hybrid model are given in the following tables. The binary decision variables used to represent a particular mapping generated by the EA are shown in Table 4.1. The specifications of the FPGA platform that are needed for the area and buffer objectives are listed in Table 4.2. Table 4.3 summarizes the mapping and input graph specific variables.

| DECISION VARIABLES | DESCRIPTION |
|---|---|
| $A_{ij}$ | Binary variable which is set to<br>= 1 if actor $i$ is mapped to $PE_j$<br>= 0 otherwise |
| $alloc_j$ | Binary variable which is set to<br>=1 if $\Sigma A_{ij} > 0$ ( $i \in$ set of actors in the graph)<br>= 0 otherwise |
| *Buffer_space* | This value lies in the range of [1, *Buffer_max*]. It denotes the number of tokens on each edge (in case of distributed memory) or total number of tokens on all edges(in case of shared memory), as described below. |

Table 4.1: Decision variables for mapping problem.

Note that we assume that virtual source and sink actors are added to the graph to serve as predecessor and successor, respectively, of all other actors. The size of a graph $I$ is determined by the total number of actors or nodes in it. This number is inclusive of the virtual source and virtual sink nodes in input design model of this work. Without loss in generality, we assume that actor 0 is the virtual source and actor $I$-1 is the virtual sink.

| FPGA VARIABLES | DESCRIPTION |
|---|---|
| *BRAM_MAX* | Total number of block RAMs (BRAM) available on the FPGA |
| *DSP_MAX* | Total number of digital signal processors (DSP) available on the FPGS |
| *SLICE_MAX* | Total number of slices available on the FPGA |
| *IP_Types* | Types of predefined IPs available on the FPGA e.g. FIR, FFT etc. |
| $BRAM_p$ | Total number of BRAMs needed by PEs of IP type $p$ |
| $DSP_p$ | Total number of DSPs needed by PEs of IP type $p$ |
| $SLICE_p$ | Total number of slices needed by PEs of IP type $p$ |
| *Area_BRAM* | Units of area required by each BRAM on the FPGA |
| *Area_DSP* | Units of area required by each DSP on the FPGA |
| *Area_slice* | Units of area required by each slice on the FPGA |

Table 4.2: Variables of the target FPGA model.

| MAPPING VARIABLES | DESCRIPTION |
|---|---|
| $J$ | Total number of Processing Elements (PEs) |
| $I$ | Total number of actors/nodes in the graph |
| $II_j$ | Initiation interval of $j^{th}$ PE (each PE is individually pipelined) |
| $d_{ij}$ | Execution time of actor i on processor j |
| $n_i$ | Number of firings of actor $i$ in each periodic iteration (repetition vector) |
| $o_{i1i2}$ | Initial tokens on edge between actors $i_1$, $i_2$ |
| $\alpha_i$ | IP type of actor $i$ |
| $\beta_j$ | IP type of processor $j$ |
| $\gamma_{ij}$ | Binary decision variable for IP matching variable between actor $i$ and $PE_j$<br>= 1 if $\alpha_i=\beta_j$<br>= 0 otherwise |

Table 4.3: Mapping model variables.

## 4.2. Encoding

In order to incorporate the target and mapping model described above, the original binary-coded genetic algorithm used in [11] is modified. The set of chromosomes, which, in EA terminology, encode the possible sets of values for every decision variable, is modified to include additional chromosomes for buffer space. Genomes, i.e. the collection of all chromosomes are rejected if the selected mapping requirements exceed the maximum number of slices, BRAMs and DSPs available in the model.

In addition to IP type compatibility, for a chromosome to be valid the following constraints need to be satisfied. If they are not then the chromosome is rejected and the scheduler is not called for the selected mapping:

$$\sum_j alloc_j * DSP_p < DSP\_MAX$$

$$\sum_j alloc_j * BRAM_p < BRAM\_MAX + Buffer\_space$$

$\sum_j alloc_j * SLICE_p \ < \ SLICE\_MAX$

where

- $j$ is a processing element belonging to range of [0, $J$-1]
- $p$ is the IP type of $PE_j$ (i.e. equal to $\beta_j$)
- *Buffer_space* is the total buffer space in case of a shared memory model or the sum over the buffer space for all edges in case of a distributed memory model.

In each generation the EA selects a value of the buffer space from its possible range and provides it to the scheduler as an upper bound constraint for that particular mapping. The overall possible range of buffer chromosomes is [1, *Buffer_max*] where *Buffer_max*, is defined as follows:

- In case of a shared memory, *Buffer_max* is the maximum of the sum of tokens present at all edges at any given instance. It is given as

  $Buffer\_max = \Sigma_e \ Repetition\_count\_source * Tokens\_produced\_on\_e$,

  where $e$ belongs to set of all possible edges in the dataflow graph
- In case of distributed memory i.e. each edge having a separate buffer space, the number of chromosomes for buffer spacing is the same as the number of edges and the Buffer_max value for a chromosome representing a particular edge is given as:

  $Buffer\_max_e = \ Repetition\_count\_source * Tokens\_produced\_on\_edge\_e$,

  where $e$ belongs to set of all possible edges in the dataflow graph

### 4.3. Objective Functions

As mentioned earlier, the EA generates a mapping corresponding to an assignment of actors to PEs (decision variable $A_{ij}$) and a selection of buffer sizes. Buffer sizes and mappings are then passed as constraints into the scheduler, which determines an optimized latency. Independently, the EA computes the period and area for the given mapping. With these, the EA drives the search towards the Pareto front for each population in all the generations such that period, area and latency are minimized. The computation of period and area for a given mapping is as follows:

- **Period Objective:**

For a given mapping, the period of the overall parallel schedule is decided by the critical processor i.e. the one that takes the longest time to finish executing one iteration of all the actors mapped onto it. The period determines the throughput of the overall graph, which is defined as the inverse of one iteration period. Hence, the period can be represented as

$$Period = \max_j \sum_I A_{ij} \gamma_{ij} ( n_i II_j )$$

Since PEs are individually pipelined, it is assumed that all actors mapped to a processor $j$, irrespective of their execution times, will successively be started after $II_j$ time units.

- **Area Objective:**

The overall area for a given mapping is computed as follows, where $p$ is the IP type of $j^{th}$ processor:

$$Area = (Area\_DSP * \sum_j alloc_j * DSP_p + Area\_slice * \sum_j alloc_j * SLICE_p + Area\_BRAM * (\sum_j alloc_j * BRAM_p + Buffer\_space)$$

# Chapter 5 Scheduling Algorithms

The two-stage hybrid heuristic for tackling the combined mapping and scheduling problem has already been introduced in the previous two chapters, along with the specific details of the genetic algorithm used in the first stage for mapping and binding. This chapter covers the crux of the thesis work, wherein the scheduling problem for a given mapping (from the GA) is dealt with. The schedules generated as part of the techniques discussed in this chapter attempt to minimize latency. *Latency* is here defined as time difference between the start of $n^{th}$ iteration of the source and the end of the $n^{th}$ iteration of the sink in the periodic schedule.

In Section 5.1, the ILP-based solution described in Jing et al. [11] is first extended from SDFGs to CSDFGs. Extension to CSDFG, however, increases the design complexity as a result of which run times of ILP-based solutions increase exponentially. To counter this increase in complexity, an optimized ILP solution is presented in Section 5.2 for SDF graphs. It significantly reduces the run time while producing the same optimal solutions as the original ILP from Section 5.1. The run time is further reduced considerably by developing a list scheduling heuristic that completely replaces the ILP-based scheduling model in the two-stage heuristic. The list scheduler is described in Section 5.3. There is a tradeoff between run time and optimality of the solution, but as will be shown later, the list scheduling heuristics developed as part of this thesis produce solutions that remain within a 10% optimality gap.

## 5.1. CSDF ILP Model

Cyclostatic dataflow graphs are extensions of synchronous dataflow graphs that are more versatile and expressive and are capable of modeling a wider range of multi-rate real-time streaming applications. This section discusses the ILP-based

model for scheduling CSDF graphs. Additional semantics needed for representing CSDFGs are discussed in Section 5.1.1. Section 5.1.2 describes the constraints that are used in the ILP model for finding a valid optimal schedule for a given mapping. As mentioned earlier, the dataflow graph at the input to the scheduler is assumed to live and consistent. The model as presented here assumes a shared buffer space for the complete graph and attempts to minimize shared buffer size along with minimizing latency. Note, however, that a buffer size objective in the ILP formulation can equally be turned into a simple buffer constraint while minimizing latency only.

### 5.1.1. ILP Formulation

The scheduler takes the mapping and the repetition vector of the dataflow graph as inputs. Most of the input parameters were already described in Chapter 3 and Chapter 4. The parallel schedule solved by the ILP contains a start-up phase followed by one period of the stable phase. Definitions of *startup* and *stable phase*s are similar to those in [11].

In this model separate decision variables have been added for different phases of the same actor. For each actor *i*, its phases are denotes by *x*. So if an actor *i* fires its $m^{th}$ time (counting from zero) then its phase *x* will be given by (*m* modulo $\phi_i$). Table 5.1 shows the list of decision variables used in this ILP formulation for the CSDGs.

Figure 5.1 shows a simple example of a CSDFG to illustrate the scheduling problem and the meaning of each of the decision variables. In this example, actors *A0* and *A5* have been added to the graph as virtual source and virtual sink, respectively. Actor *A1* has two phases while all other actors have a single phase. The repetition vector of (*A0, A1, A2, A3, A4, A5*) is (1,1,2,1,1,1). The values shown in square brackets denote the execution times of each of the phases of a given actor for a given mapping. Also, the values on the edges represent the token consumption and production rates. For example, on edge (*A1, A2*), actor *A1* produces 2 tokens in its first phase and 0 tokens in its second.

| DECISION VARIABLES | DESCRIPTION |
|---|---|
| $S_{ix}(t)$ | Number of times actor $i$ started its execution in its phase $x$ (integer) until time instance $t$ ($t$ inclusive) |
| $E_{ix}(t)$ | Number of times actor $i$ ended its execution in its phase $x$ (integer) until time instance $t$ ($t$ exclusive) |
| $E'_{ix}(t)$ | Number of times actor $i$ virtually ended its execution in its phase $x$ (integer) until time instance $t$ ($t$ exclusive). |
| *start(t)* | indicates beginning of stable periodic phase starting at time $t$+1 |

Table 5.1: Decision variables used in CSDFG ILP formulation.

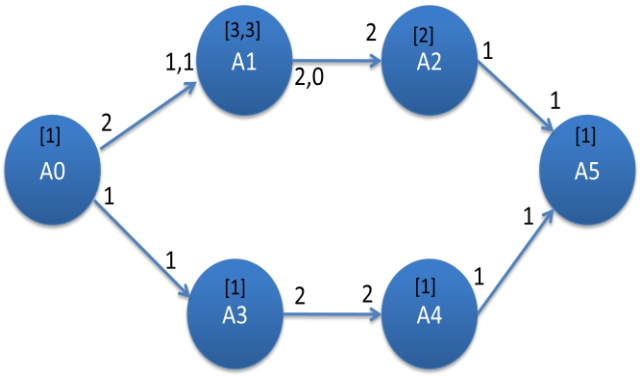


Figure 5.1: Example of CSDFG.

An architecture comprising of three PEs is assumed, such that actors *A0* and *A5* are mapped to *PE1*, *A1* and *A3* to *PE0*, and *A2* and *A4* to *PE2*. It is assumed that *PE0* is also pipelined with an initiation interval of 2 time units. The *Period* of the overall graph is assumed to be 6 time units.

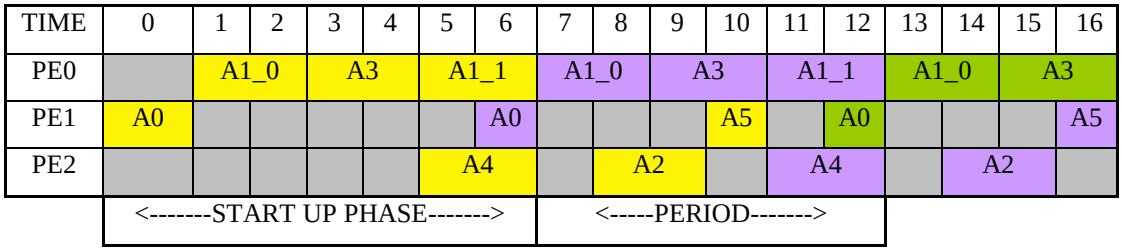


Figure 5.2: Example schedule.

| **Decision Variables** | **Time Units** | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
| **Start** | 0 | | | | | | **1** | 0 | | | | | | | | | |
| **$S_{A1,0}$** | 0 | 1 | | | | | | 2 | | | | | | 3 | | | |
| **$E_{A1,0}$** | 0 | | | | | 1 | | | | | 2 | | | | | 3 | |
| **$E'_{A1,0}$** | 0 | | | | | 1 | | | | | 2 | | | | | 3 | |

Figure 5.3: Decision variables for given example.

Figure 5.2 shows the schedule generated by the ILP solver for the given scenario. Different colors indicate different iterations. As can be seen above, the schedule comprises a start-up phase that precedes the periodic phase. On *PE0*, the first phase (or $0^{th}$ phase as per the numbering notation used) of actor *A1* (*A1_0*) begins its execution at time instance 1 i.e. $S_{A1,0}(1) = 1$. Since *PE0* is pipelined with an initiation interval of 2 time units, actor *A3* begins its execution at time instance 3 even though actor *A1_0* has not yet ended its execution. An illustration of the decision variables is shown in Figure 5.3. The decision variable *start* is high at time instance *t*=6, indicating that starting at *t*=7, the schedule has reached a periodic phase.

- $S_{A1,0}(1) = 1$, indicating the beginning of execution the first execution of *A1_0*. It increments to 2 when the next firing begins for the given phase at time instance 7, and so on and so forth.
- $E'_{A1,0}(3) = 1$, indicating that $II_{PE0}$ time units have elapsed since the beginning of actor *A1_0*'s first execution and that, consequently, the next ready actor can begin its execution on the PE. Similarly it gets incremented at time instances 9 and 15.

- $E_{A1,0}(4)$ =1, indicating the end of *A1_0*'s first execution or firing. It is at this time instance that tokens produced by *A1_0's* firing are added onto the edge (*A1*, *A2)*. Similarly it gets incremented at time instances 10 and 16.

### 5.1.2. ILP Constraints

This section describes the ILP problem formulation for finding a schedule for CSDF graphs under the following given constraints:

- Counting processes with unit increments

The decision variables are unit-counting variables, i.e. they are incremented by one time unit each time an actor begins or ends its execution:

$$0 \leq S_{ix}(\mathrm{t}+1) - S_{ix}(\mathrm{t}) \leq 1 \quad (1)$$

$$0 \leq E_{ix}(\mathrm{t}+1) - E_{ix}(\mathrm{t}) \leq 1 \quad (2)$$

$$0 \leq E'_{ix}(t+1) - E'_{ix}(t+1) \leq 1 \quad (3)$$

The above set of equations hold true for $i \in$ set of actors and their phases $x$ in the range of $[0, \phi_i)$.

- Initialization

Before the scheduler starts finding a schedule, it is assumed that none of the actors except the virtual source have begun their execution. The virtual source is scheduled at time instance 0 during the initialization itself, i.e. $S_{00}(0)$=1. For the rest of the actors the initialization constraints is given as

$$S_{ix}(1) \leq 1 \quad (4)$$

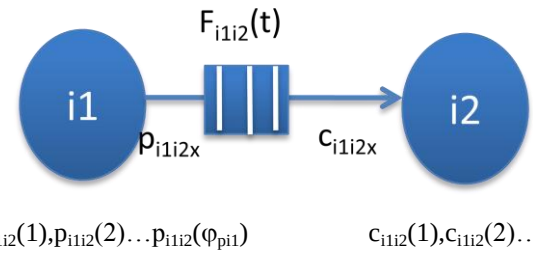


Figure 5.4: Edge in a CSDFG.

- Firing rule (precedence)

As mentioned earlier, an actor in a dataflow graph begins its execution (fires) only when it has sufficient tokens on all its input edges. Figure 5.4 shows an edge $i_1i_2$, where all variable definitions are same as those defined in Chapter 3.

The total number of tokens produced by actor $i_1$ on edge $i_1i_2$ till time instance *t*, is proportional to the total number of times it ended its execution in all its phases and is given by:

$$X_{i1i2} = \sum_x p_{i1i2x} * E_{i1x}(t) + o_{i1i2} \quad (5)$$

The total number of tokens consumed by actor $i_2$ on edge $i_1i_2$ till time instance *t*, is proportional to the total number of times it started its execution in all its phases and is given by:

$$Y_{i1i2x} = \sum_x c_{i1i2x} * S_{i2x}(t) \quad (6)$$

*(Time instance t is inclusive in both Equations 5 & 6)*

For actor $i_2$ to begin its execution at time instance *t*, it should have a sufficient number of tokens on the edge, hence the following condition must hold true for the same:

$$X_{i1i2} - Y_{i1i2} \geq 0 \quad (7)$$

The above constraint must hold true for all edges ($i_1i_2$) of the given graph.

- Execution time

Actor *i* beginning its execution in its $x^{th}$ phase will end its execution after $d_{ijx}$ time units:

$$S_{ix}(t) = \sum_j A_{ij}\gamma_{ij}E_{ix}(t + d_{ijx}) \quad (8)$$

Also the relationship between $S_{ix}(t)$ and $E'_{ix}(t)$, explained by example in Figure 5.1, is given by:

$$S_{ix}(t) = \sum_j A_{ij}\gamma_{ij}E'_{ix}(t + II_j) \quad (9)$$

The above two conditions should hold true for all $i \in$ set of actors and their phases $x$ in the range of $[0, \phi_i)$

- Unique processor mapping

Each actor can be mapped to a single processor only. So the following condition should hold true for all actors:

$$\sum_j A_{ij}\gamma_{ij} = 1 \tag{10}$$

- Sequential execution on each processor

Actors mapped to the same processor cannot begin their execution simultaneously. There has to be a minimum gap of one initiation interval, equal to $II_j$ time units, of the corresponding processor $j$. Hence the following constraint should be satisfied for each processor $j \in$ set of PEs:

$$\sum_i A_{ij}\gamma_{ij} \sum_x (S_{ix}(t) - E'_{ix}(t)) \leq 1 \tag{11}$$

- Unique start time

There is only one stable phase in the considered time frame:

$$\sum_t start(t) = 1, \tag{12}$$

where $0 \leq t \leq T$ is the total time frame taken by the scheduler to reach a periodic schedule, i.e. the sum over startup and periodic phases.

- Precedence of phases for each actor

Each actor executes its phases cyclically and sequentially, i.e. unless the 1st phase of actor i has ended its execution, the 2nd phase cannot be started, etc. The constraint needs to be satisfied for all actors $i$, where $(x+1)$ denotes the succeeding phase of $x$ for actor $i$:

$$0 \leq E_{ix}(t) - S_{i(x+1)}(t) \leq 1 \quad (13)$$

for all $i \in$ set of actors and their phases $x \in [0, \phi_i)$

Constraint 13 ensures that a later phase ($x$+1) can never start before the preceding phase ($x$) has ended. Similarly, it needs to be ensured that the number of times actor $i$ begins its execution in phase $x$ is greater than or equal to the number of times it begins its execution in phases $x$+1, $x$+2, $x$+3… ($\phi_i$-1). Constraint 14 represents a set of relationships that needs to be satisfied for each of the phases:

$$0 \leq S_{ix}(t) - S_{i(x+1)}(t) \leq 1 \quad (14)$$

for all $i \in$ set of actors and their phases $x \in [0, \phi_i)$

- Periodicity

The periodic phases contain exactly one iteration of the CSDFG. Hence if $W_i(t)$ is defined as the duration for which actor $i$ has executed (all phases included) till time instance $t$:

$$W_i(t) = \sum_{0 \leq \tau \leq t} \sum_x (S_{ix}(\tau) - E_{ix}(\tau)) \quad (15)$$

Then the following condition must hold true for all actors to ensure periodicity of the schedule [11]:

$$W_i(T) - \sum_t W_i(t) * start(t) = \sum_j A_{ij}\gamma_{ij} \sum_x n_i * d_{ijx} \quad (16)$$

### 5.1.3. ILP Objectives and Cost Function

As mentioned earlier, the ILP based scheduling model attempts to find a schedule that minimizes latency and overall buffer space. The objective functions are given as below:

- **Buffer usage**

As shown in Fig. 6, at edge ($i_1$,$i_2$), if $i_1$ executes in its $x^{th}$ phase at time $t$ then the total number of tokens produced until $t$ is given by

$$P_{i1i2}(t) = (\sum_t \sum_x E_{i1x}(t) * p_{i1i2x}) + p_{i1i2x} \quad (17)$$

$$= \sum_t \sum_x S_{i1x} * p_{i1i2x} \quad (18)$$

(time *t* is inclusive the above relation)

At edge ($i_1$,$i_2$) the total number of tokens consumed by $i_2$ until time instance *t* is given by:

$$C_{i1i2x}(t) = \sum_t \sum_x E_{i2x}(t) * c_{i1i2x} \quad (19)$$

(time *t* is exclusive the above relation)

Hence, the total accumulated tokens until *t* on edge ($i_1$,$i_2$), denoted by $F_{i1i2}(t)$, is given by:

$$F_{i1i2}(t) = P_{i1i2}(t) - C_{i1i2}(t) + o_{i1i2} \quad (20)$$

Substituting the values of $P_{i1i2}(t)$ and $C_{i1i2}(t)$, $F_{i1i2}(t)$ is given as:

$$F_{i1i2}(t) = \sum_t \sum_x S_{i1x}(t) * p_{i1i2x} - \sum_t \sum_x E_{i2x}(t) * c_{i1i2x} + o_{i1i2} \quad (21)$$

Based on the above equation, which denotes the buffer space needed by edge($i_1$,$i_2$) at a given time instance, the maximum buffer space needed by edge ($i_1$,$i_2$) in the time span *T*, including *startup* and *periodic phases*, is given as:

$$Buffer_{i1i2} = Maximum\ (F_{i1i2}(0), .., F_{i1i2}(t), .., F_{i1i2}(T-1)) \quad (22)$$

Hence the total buffer size required by the graph is the sum over the maximum buffer sizes required by all the edges. It can be represented as follows:

$$Buffer\_size = \sum_{all\ edges} Buffer_{i1i2} \quad (23)$$

- **Latency**

As mentioned in Jing et al. [11], latency is defined as the summation of the duration between the first start of the virtual source and first start of the virtual sink

during the period, plus the execution time of the virtual sink and the difference between their iteration numbers. It is given as:

$$Latency = \sum_t \big(U(t) - V(t)\big) + \sum_j A_{Ij} * d_{Ij} + \big(S_0(T) - S_I(T)\big) * Period, \quad (24)$$

where *Period* = T - $\Sigma_t$ t*start(t) is the period of the graph and *U(t)* and *V(t)* are binary variables that signify the beginning of the first executions of the virtual source and virtual sink, respectively.

The overall objective function of this ILP formulation is a linear combination of latency and buffer space. Linear weights may be assigned to each of the two objectives to give higher or lower preference to one over the other, i.e.

$$Minimize[\lambda_1 * Latency + \lambda_2 * Buffer\_size], \quad (25)$$

where $\lambda_1, \lambda_2$ are the linear weights assigned to the two objectives. Note that instead of including it in the cost function to minimize, buffer size can simply be constrained by an upper bound (of *Buffer_space* given by the EA) instead.

## 5.2. Improved ILP Model

As mentioned earlier, the ILP model presented in the previous section requires a startup phase in addition to the periodic phase. This implies that the formulation has to traverse additional time units, and hence decision variables, until it reaches a stable periodic phase. The increase in the time frame to be considered negatively impacts the run time of the ILP solver. In this section, we present an optimization that is based on the approach in [3] and thereby gets rid of the startup phase by establishing a periodic relationship between the source and sink of each edge before beginning the scheduling. The ILP formulation in Govindrajan's work [3] does not consider any hardware constraints. We extend this approach and model hardware constraints in the formulation presented in this section using the concept of a modular reservation table (MRT) [6]. Compared to the ILP presented in the previous section, this modified ILP has a significantly reduced runtime by almost a factor of 10, while still maintaining the same optimality in the solutions.

This formulation has been implemented for SDFGs in this work, but it is equally applicable for and can be easily extended to CSDFGs following the formulation presented in Section 5.1. In the following, Section 5.2.1 describes the alternate definitions needed for this formulation on top of the variables described in Chapter 4. Section 5.2.2 and Section 5.2.3 then elaborate upon the constraints and objective functions of the formulation.

### 5.2.1. Modified ILP Formulation

This formulation takes in the same input variables as described in Chapter 3 and Section 4.1. In contrast to the formulation in Section 5.1, however, the improved model is based on a different set of decision variables listed in Table 5.2. In this model, counting variables are replaced with binary variables for each distinct instance of an actor execution in the overall periodic repetition vector

| DECISION VARIABLES | DESCRIPTION |
|---|---|
| $S_{ik}(t)$ | Binary variable, equal to 1 when $k^{th}$ instance of actor *i* begins execution |
| $E_{ik}(t)$ | Binary variable, equal to 1 when $k^{th}$ instance of actor *i* ends execution |
| $MRT_j(t)$ | A two dimensional Modulo Reservation Table whose rows represent processing elements in the input model and the number of columns is equal to the number of time slots in the period for the given graph. It is equal to 1 at time instances *t* when $PE_j$ is busy, i.e. an actor mapped to it has started its execution but hasn't completed yet. |

Table 5.2: Decision variables for the modified ILP formulation.

### 5.2.2. Modified ILP Constraints

This section describes the constraints for the modified ILP model. They are mainly based upon those described in Section 4.2.2, and the modifications have been highlighted here. The maximum time that formulation will be modeled for is *T*, which in this case corresponds to the maximal time for completion of one iteration of the graph assuming all actors are mapped to a single processor. Just like the previous ILP formulation, this model also takes in the Period as input constraint from the GA. It tried to minimize *Latency* and *Buffer_space* requirements of the graph in its objective function. A shared memory model is assumed in this case, similar to that in Section 5.1.

- Execution for one iteration only

This formulation models only a single iteration without a startup phase to produce an overlapped periodic schedule. Hence, each instance of every actor is executed exactly once in the time duration modeled by the formulation. This condition is expressed as the following constraint:

$$\sum_t E_{ik}(t) = 1 \tag{26}$$

for all $i \in$ set of actors and their instances $k \in [0, n_i)$

- Execution time

The relationship between the end and the start times of firing of $k^{th}$ instance of actor $i$ same as constraint 8. It can therefore be written in a similar way as:

$$S_{ik}(t) = \sum_j A_{ij} * \gamma_{ij} * E_{ik}(t + d_{ij}) \tag{27}$$

for all $i \in$ set of actors and their instances $k \in [0, n_i)$

- Unique processor mapping

Just like the previous formulation, this modified ILP model assumes that an actor is uniquely mapped to a single processor. It is same as constraint 10.

$$\sum_j A_{ij}\gamma_{ij} = 1 \tag{28}$$

- Sequential execution on each processor

A dependent binary decision variable $U_{ik}(t)$ is introduced, which is set to 1 while the $k^{th}$ instance of actor $i$ is executing, i.e. has started its execution but not completed yet. It is set back to 0 on the completion of the execution:

$$U_{ik}(t) = \sum_{0 \le \tau \le t} (S_{ik}(\tau) - E_{ik}(\tau)) \tag{29}$$

for all $i \in$ set of actors and their instances $k \in [0, n_i)$

Each entry in the MRT at a given time instance $t$ signifies if the corresponding PE is available or not. If an actor $i$ mapped to processor $j$ is executing at time instance $t$, then $PE_j$ is said to be busy at that time. All the entries in the MRT can, therefore, be populated as follows using the dependent decision variable introduced above:

$$MRT_j(t) = \sum_{\tau = t, (t + Period) \ldots T} \sum_i \sum_k A_{ij} * \gamma_{ij} * U_{ik}(\tau) \tag{30}$$

for $t \in [0, Period)$

The following constraint then ensures that only a single actor can be executing on a PE at a given time:

$$MRT_j(t) \leq 1 \quad (31)$$

for all $t \in [0, Period)$ and $j \in$ set of PEs ($J$).

- Precedence of instances for each actor

This constraint is similar to that described in Equations 13 and 14 for establishing precedence between different phases of the actors. In this case $k^{th}$ instance of actor should always precede its $(k+1)^{th}$ instance, and so on and so forth. Hence the following two conditions should always hold true for different instances of each actor in the graph:

$$0 \leq E_{ik}(t) - S_{i(k+1)}(t) \leq 1 \quad (32)$$

$$0 \leq S_{ik}(t) - S_{i(k+1)}(t) \leq 1 \quad (33)$$

for all $i \in$ set of actors and their instances $k \in [0, n_i\text{-}1)$

- Firing rule (precedence constraint)

Precedence constraint has been modified in this ILP model based upon the precedence relationships described in Govindrajan's work [3]. Consider the edge $i_1i_2$ shown in Figure 5.5. Assume actor $i_2$ started its $k_2^{th}$ firing in the $j_2^{th}$ iteration of the graph at time $t$. The total firings of the sink ($i_2$) can then be represented as ($j_2 * n_{i2} + k_2 + 1$). It is taken as ($k_2$+1) since the numbering convention in this work starts from 0 so when an actors is in its 2nd firing, it implies it has fired 0th, 1st and 2nd times i.e. total 3 times. Therefore, the total number of tokens consumed by $i_2$ till time $t$ can be represented as:

$$Input_{i1i2} = (j_2 * n_{i2} + k_2 + 1) * c_{i1i2} \quad (34)$$

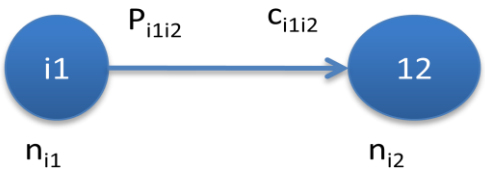


Figure 5.5: Edge in SDFG.

Assuming actor $i_1$ completed its $k_1^{th}$ instance in the $j_1^{th}$ iteration of the graph by time $t$. The total firings of the source ($i_1$) can then be represented as ($j_1$*$n_{i1}$ + $k_1$ + 1) and hence the total tokens produced by $i_1$ on edge $i_1i_2$ can be represented as:

$$Output_{i1i2} = (j_1 * n_{i1} + k_1 + 1) * p_{i1i2} \quad (35)$$

Since an actor in a dataflow graph begins its execution only if it has a sufficient number of tokens on all its input edges, the following condition must hold true, such that the total number of tokens consumed is less than the total number of tokens produced plus the number of initial tokens on the edge:

$$Input_{i1i2} \leq Output_{i1i2} + o_{i1i2} \quad (36)$$

For computing a lower bound on the possible execution difference that the source needs to be ahead of the sink, an equality is assumed in Equation 36. Note that depending on the number of initial tokens, the lower bound can be negative and that the ILP solver will in either case be able to schedule with a higher execution difference between sink and source as per its precedence constraint defined later. Substituting the expressions for $Input_{i1i2}$ and $Output_{i1i2}$ from Equations 34 and 35 respectively into Equation 36:

$$(j_2 * n_{i2} + k_2 + 1) * p_{i1i2} = (j_1 * n_{i1} + k_1 + 1) * c_{i1i2} + o_{i1i2}$$

$$(j_1 * n_{i1} + k_1) = \frac{((j_2 * n_{i2} + k_2 + 1) * c_{i1i2} - o_{i1i2} - p_{i1i2})}{p_{i1i2}} \quad (37)$$

Hence, for the sink actor $i_2$ to be able to fire its $k_2^{th}$ instance in its $j_2^{th}$ iteration, the minimum number of instances and full iterations the source $i_1$ to be completed can be computed as:

$$k_1 = \left\lceil \frac{(j_2 * n_{i2} + k_2 + 1) * c_{i1i2} - o_{i1i2} - p_{i1i2}}{p_{i1i2}} \right\rceil \%n_{i1} \quad (38)$$

And

$$j_1 = \left\lfloor \frac{\left\lceil \{(j_2 * n_{i2} + k_2 + 1) * c_{i1i2} - o_{i1i2} - p_{i1i2}\} / p_{i1i2} \right\rceil}{n_{i1}} \right\rfloor \quad (39)$$

The dataflow graph's property states that in one iteration the number of tokens produced by the source at an edge should be equal to the number of tokens consumed by the sink so that after each iteration, initial state of the graph is restored. Hence in this case of edge $i_1i_2$, it requires that the following condition should hold true:

$$c_{i1i2} * n_{i2} = p_{i1i2} * n_{i1} \quad (40)$$

Using Equation 40 in Equations 38 and 39, the following relations can be established:

$$j_1 = j_2 + \left\lfloor \frac{1}{n_{i1}} \left\lceil \frac{(k_2 + 1) * c_{i1i2} - o_{i1i2} - p_{i1i2}}{p_{i1i2}} \right\rceil \right\rfloor \quad (41)$$

From the above Equation 41, denoting $j_{lag}$ as

$$j_{lag} = \left\lfloor \frac{1}{n_{i1}} \left\lceil \frac{(k_2 + 1) * c_{i1i2} - o_{i1i2} - p_{i1i2}}{p_{i1i2}} \right\rceil \right\rfloor \quad (42)$$

So Equation 41 can be represented as

$$j_1 = j_2 + j_{lag}, \quad (43)$$

where $j_{lag}$ can be either negative or zero. A zero value of $j_{lag}$ would imply that $(j_2,k_2)^{th}$ firing of $i_2$ begins after $k_1^{th}$ firing of $i_1$ in the same iteration. A negative value of $j_{lag}$ gives the iteration distance between the corresponding firings of the two actors. As can be seen from Equation 42, $j_{lag}$ is not dependent on any of the run time variables so it can be pre-computed for each pair of source and sink actors. Similarly using Equation 40, Equation 38 can be re-written as:

$$k_1 = \left\lceil \frac{(k_2 + 1) * c_{i1i2} - o_{i1i2} - p_{i1i2}}{p_{i1i2}} \right\rceil \%n_{i1} \quad (44)$$

The relationships between firings of source and sink nodes can then be used for formulating the precedence constraint. It is given as follows:

$$\sum_t tS_{i2k2}(t) - \sum_t tS_{i1k1}(t) \geq \ Period * j_{lag} + \sum_j A_{ij}\gamma_{ij}d_{ij} \qquad (45)$$

for all source and sink pairs in the given graph.

A separate periodicity constraint is not required in this formulation since it is already folded into the precedence constraint described above. The above formulation of precedence constraint has been described in detail along with examples in Govindrajan's work [3].

### 5.2.3. Modified ILP Objective Functions

Latency and buffer objective functions are formulated similar to 5.1.3.

### 5.3. List Scheduling Heuristic

In the previous Sections 5.1 and 5.2, ILP-based models for finding optimal schedules of SDFGs and CSDFGs under given throughput-latency-cost objectives have been presented. However, the complexity of ILP-based models increases exponentially with an increase in the graph and hence problem size. In order to combat this exponential increase in complexity, heuristics can be adopted. Even though solutions of heuristics are not optimal, there is a significant reduction in problem complexity, which is reflected in lower run times. As described in Chapter 2, heuristic-based scheduling approaches have been studied extensively in the past both in compiler and high-level synthesis domains. As part of this thesis, we have developed novel heuristics that combine static scheduling of SDFGs with iterative modulo scheduling algorithms from the compiler domain [7] for finding latency-optimized schedules under given throughput and resource constraints. This heuristic, which follows a list scheduling approach using height-based priorities, can be plugged into the two-stage heuristic discussed in Chapter 3 in place of the previously discussed ILP models. Such list scheduling heuristics can thereby find near-optimal minimum latency schedules for a given throughput target in linear time complexity based on a pre-defined mapping of an SDFG graph onto a heterogeneous processor platform as determined by the GA.

The objective of modulo scheduling is to find a schedule for one iteration of a dataflow graph, such that when this same schedule is repeated at regular intervals (initiation interval or period), no inter- and intra-iteration dependencies are violated, and no resource usage conflicts arise between nodes in either the same or distinct iterations. Dependencies between two instances of the same node in different iterations are referred to as *inter-iteration dependencies*, while those existing between different nodes in the same iteration are referred to as *intra-iteration dependencies*. The scheduler keeps a track of resource usage via a resource reservation table wherein each row corresponds to a processing element (PE) and each column corresponds to one time cycle. Hence, scheduling of a node $x$ on processing element $PE_x$ at time $t$

corresponds to a [$PE_x$, $t$] entry in the table. This reservation table, when modified for adherence to modulo constraints imposed by a periodic, modulo scheduling is called a *Modulo Reservation Table* (MRT) [6]. Consequently, scheduling of a node on $PE_x$ at time $t$ corresponds to a [$PE_x$, *(t* modulo *Period)*] entry in an MRT.

A modulo-constrained version of list scheduling can, however, very often lead to a dead-end state corresponding to a partial schedule where the scheduler is unable to find an available slot of enough consecutive entries for the selected node on the corresponding PE even though enough free resources are overall available. In such a scenario, the backtracking nature of modulo scheduling algorithms allows them to start unscheduling conflicting (previously scheduled) nodes. Search continues until a legal schedule is found for the given graph. The backtracking ability of these algorithms increases the probability of finding a legal schedule as compared to non-backtracking list scheduling algorithms.

However, back-tracking is expensive and not always successful, e.g. when running into an endless cycle in which nodes keep unscheduling each other. In order to further optimize the algorithm, we have developed a concept of mobility-based rescheduling, which has been added to our algorithm. Such rescheduling is attempted before resorting to backtracking on encountering a dead-end state. As will be shown later in the experimental results (Chapter 6), mobility-based rescheduling in addition to backtracking further increases the probability of finding a legal solution and hence the optimality of the algorithm. This algorithm is presented in detail in the sections to follow.

### 5.3.1. List Scheduling Formulation

The input parameters to the heuristic are the same as for the ILP model described in Section 4.1. The list scheduling model assumes distributed buffer space, i.e. separate memory for each edge. Buffer sizes for the edges are generated by the corresponding chromosomes in the EA and taken as input constraint by the list scheduler i.e. the scheduler doesn't attempt to minimize buffer space. Instead it only

ensures that actors are scheduled such that at any instance of time, the total number of tokens at any edge never exceeds the allotted buffer space to that edge by the GA. Hence, the overall period of the graph and the buffer sizes for all the edges are provided as input constraints by the EA to the list scheduler. The list scheduler attempts to find a valid schedule, satisfying those constraints in addition to the given resource mapping, such that the latency of the schedule is minimized. The data structures used in the list scheduling algorithm are summarized in Table 5.3.

The list scheduler uses a height-based priority function for maintaining the ordered *ReadyList*. Higher priority is assigned to a node that has a longer critical path to the sink node of the graph. The sink node is the node that has no output edges. The height of sink node is defined as 0, and the priority assignment scheme used in this thesis is derived from [16], wherein different instances of the same actor are assigned different priorities with the first instance having the highest and the last instance having the lowest priority.

Let $Height(i_k)$ be the priority of $k^{th}$ invocation of actor $i$, $d_i$ be the execution time of the actor, and $n_i$ be the repetition count of the actor $i$. Then, the priority of a node invocation is determined by the following two equations:

$$Height_{i\_last} = Maximum\{Height_Q\} + d_i \quad (46)$$

$$Height_{ik} = Height_{i_{last}} + (n_i - k) * d_i \quad (47)$$

Where $Q$ belongs to the set of successors of node $i$, $i_{last}$ is the $n_i^{th}$ instance of node $i$ and $i_k$ is the $k^{th}$ instance of node $i$. Hence, every time a node is executed or fired, subtracting its execution time from the current height value lowers its priority. The above set of equations is executed for computing the priority of all the nodes in the graph recursively.

| DATASTRUCTURE | DESCRIPTION |
|---|---|
| *ReadyList* | List of nodes which are ready to be executed/fired |
| *UnfinishedList* | List of nodes whose execution has started but not ended yet by the current time instance |
| *FinishedList* | List of nodes whose execution ends at current time instance |
| *Current_time* | Current time instance up to which point scheduling has progressed |
| *Next_time* | Earliest time instance when a node from *UnfinishedList* will end its execution |
| $Height_i$ | Distance of node *i* from sink, which is used for determining its priority |
| $Channel_{i1i2}$ | Number of tokens on edge $i_1i_2$, with node $i_1$ being the source and node $i_2$ being the sink, at current time instance |
| $S_i(t)$ | Binary variable that is set to 1 when node *i* begins its execution at time instance *t*, otherwise set to 0 |
| $E_i(t)$ | Binary variable that is set to 1 when node *i* ends it execution at time instance *t*, otherwise set to 0 |
| $Count\_started_i$ | Integer variable that keeps a track of the number of times actor *i* has started its execution until the *Current_time*. It is incremented every time an actor is pushed into the *UnfinsihedList* and decremented if it is unscheduled during backtracking |
| $Count\_ended_i$ | Integer variable that keeps a track of the number of times actor *i* has ended its execution until the *Current_time*. It is incremented every time an actor is pushed into the *FinishedList* and decremented if it is unscheduled during backtracking |
| $MRT_j(t)$ | Two-dimensional Modulo Reservation Table for keeping track of resource usage. It is equal to 1 at time instance *t* when $PE_j$ is busy, i.e. an actor mapped to it has started its execution but has not completed yet. |

Table 5.3: Datastructures used in list scheduling algorithm.

An *Update_ReadyList* function then checks the readiness of actors for execution and updates the *ReadyList* accordingly. An actor is said to be ready for execution or firing when it has a sufficient number of tokens on all its input edges. Since the heuristic finds a schedule for only one iteration of the graph, the number of completed firings of the ready actor should also be less than its repetition count. Finally, as mentioned earlier, the scheduler takes the maximum buffer space for each edge as a constraint from the EA. So in addition to the above two conditions, an actor should also have sufficient space on all its output edge buffers before it can begin its execution. All combined, an actor $i_2$ is said to be ready for firing or execution at a given time instance t when the following conditions are satisfied:

$$Count\_started_{i2} * c_{i1i2} \leq Count\_ended_{i1} * p_{i1i2} + o_{i1i2} \quad (48)$$

$$Channel_{i2i3} + p_{i2i3} \leq Buffer\_space_{i2i3} \quad (49)$$

$$Count\_ended_{i2} < n_{i2} \quad (50)$$

Here $i_1$ belongs to the set of predecessors of actor $i_2$ (input edges), $i_3$ belongs to the set of successors of actor $i_2$ (output edges), and $Buffer\_space_{i2i3}$ is the maximum buffer space assigned to edge $i_2i_3$ by the EA.

If all the above three conditions are satisfied then actor $i_2$ is pushed into the *ReadyList* at time instance *t,* from which it will be scheduled depending on its priority and the availability of resources

An additional *Update_Channel* helper function is used to update the total tokens on all channels whenever a sink starts executing or a source ends its execution. When an actor $i_2$ begins its execution, the total number of tokens on all its input edges is updated as follows:

$$Channel_{i1i2} = Channel_{i1i2} - c_{i1i2}, \quad (51)$$

where $i_1$ belongs to the set of predecessors of actor $i_2$ (input edges).

Similarly, when actor $i_2$ ends its execution, this function updates the total number of tokens on all of its output edges as follows:

$$Channel_{i2i3} = Channel_{i2i3} + p_{i2i3}\ , \quad (52)$$

where $i_3$ belongs to the set of successors of actor $i_2$ (output edges).

Finally, an *Update_MRT* function is called to update the modulo reservation table every time an actor mapped to the PE begins its execution. If an actor *i* is scheduled at time instance *t*, the MRT of the corresponding $PE_j$ onto which it is mapped gets updated as follows:

$$MRT_j(\tau) = 1 \tag{53}$$

for all $\tau \in$ [t, t + Minimum($d_{ij}$, $II_j$)]. The same function is called during backtracking when an actor gets unscheduled. In that case, the corresponding entries of the associated PE are set to 0 instead of 1, signifying the freeing up of the resource.

### 5.3.2. Mobility-Based Scheduling

For every ready actor *i* in the *ReadyList* at time instance *t*, the scheduler tries to find a sufficient number of available contiguous time slots on the given target $PE_j$ (onto which it is mapped) so that *i* may be scheduled there and then. Since the PEs in this model are individually pipelined with initiation interval of $II_j$, actor *i* requires $d_{ij}$ or $II_j$ time slots, whichever is lower.

A *Modulo_timeslot* function, shown in Figure 5.6, is called by the core scheduler to perform this time slot assignment. This function may result in one of the following three scenarios while trying to find available slots for ready actor *i* starting time instance *t* on processor $PE_j$ onto which it is mapped:

- If the number of slots available on $PE_j$ is less than the required number of slots then this function returns a *Fail* condition, signifying that a valid schedule cannot be found for the graph.
- If sufficient contiguous time slots are available starting at a time instance greater than or equal to *t* then this value is returned and the actor is scheduled.

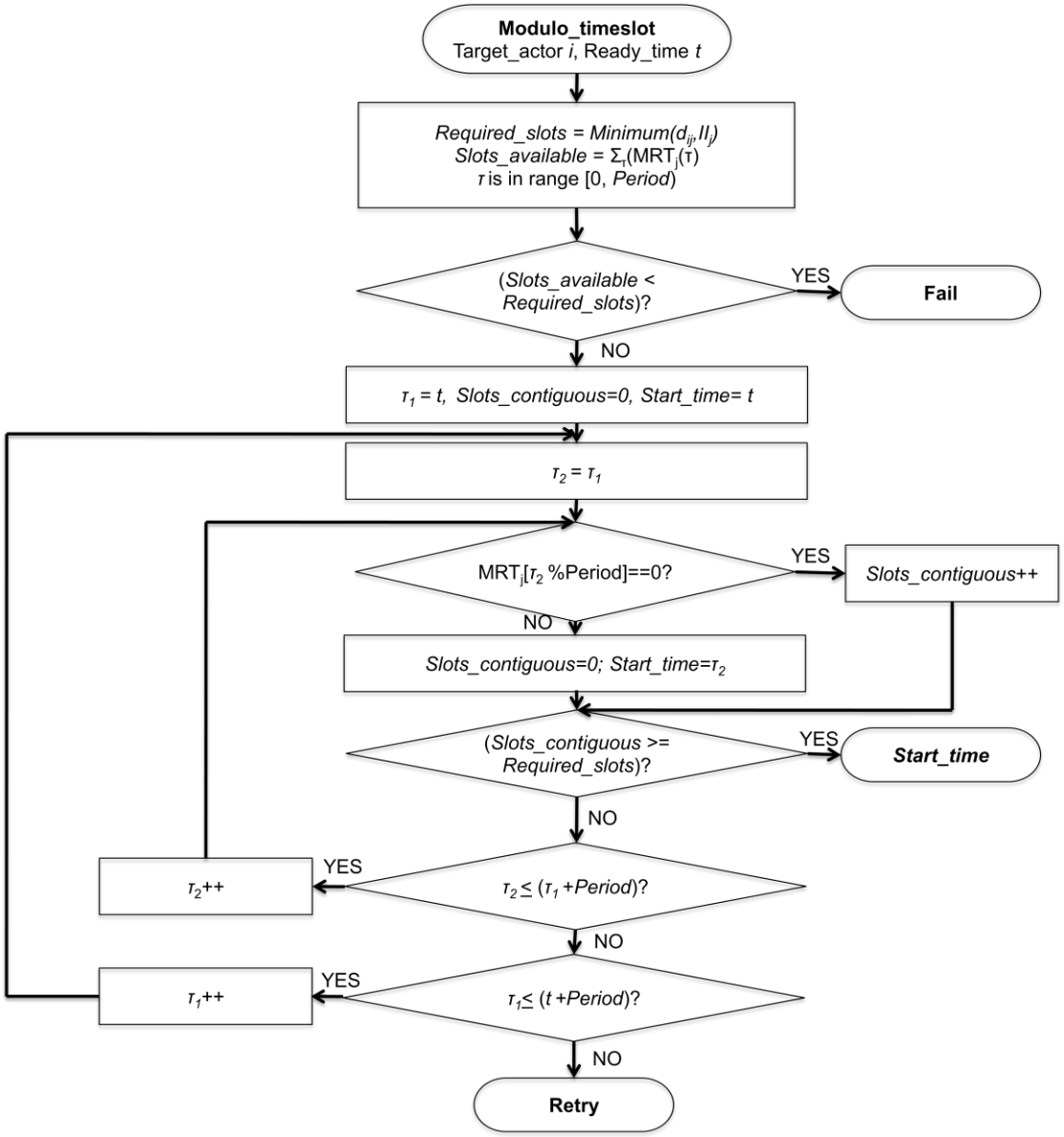


Figure 5.6: *Modulo_timeslot* function to find time instance for scheduling actor *i*.

- If sufficient slots are available, but are not contiguous then *Module_timeslot* returns a *Retry* condition, which signifies that a rescheduling of actors mapped to $PE_j$ should be attempted so as to try to create enough contiguous time slots to schedule the target actor *i*.

In this model, the firing of an actor depends not only on the availability of sufficient tokens on its input edges (i.e. its precedence constraints), but also upon the availability of resources and buffer space on its output edges. Precedence and resource availability determine the scheduling slack of an actor, i.e. the possible range of time slots into which it can be scheduled. This slack in the schedule in turn is called the *mobility* of an actor. Two types of mobilities, left or right, may be associated with an actor depending on whether the slack is present before the start or after the end of its execution. Ideally, since a list scheduler will schedule an actor as soon as all the constraints are satisfied, i.e. precedence, resource availability and buffer space, actors should only have a right mobility, if at all. However, since this scheduler incorporates backtracking, which, as will be described later, leads to unscheduling and rescheduling of actors, they may at times also have left mobility.

In our formulation, we aim to exploit actor mobility for rescheduling before expensive backtracking is attempted. Whenever the *Modulo_timeslot* function returns *Retry*, it implies that the target ready actor is prevented from being scheduled due to non-availability of required number of contiguous time slots. In these situations, a *Cleanup* function (Figure 5.7) is called. In an attempt to create sufficient number of contiguous time slots for the scheduling of target actor, the *Cleanup* function then tries to reschedule actors by exploiting their mobilities.

Since the scheduling algorithm is driven by latency minimization, creation of earliest possible contiguous time slots is desirable. A rescheduling based on left mobility has a higher probability of yielding an earlier start time for the target actor than one based on right mobility. Moving an actor out in time very often pushes the complete schedule further in time, thereby increasing the latency. Hence, as shown in Figure 5.7, the *Cleanup* function attempts rescheduling of actors based upon their left mobility first. Input to the *Cleanup* function is the target actor *i*, which even though is ready for execution at time instance *t*, is not scheduled due to non-availability of resources. The key features of the function are as follows:

- A *MappedList* is generated that comprises all actors mapped to target processor $PE_j$. It is sorted such that an actor with higher priority and hence scheduled earlier in time is considered for left mobility based rescheduling before actors scheduled later in time. An attempt to move higher priority actors back in time is made first because they in turn control the left mobility of their successors (lower priority actors).
- If a non-zero mobility value is returned by a *Left_mobility* function for a given actor then this actor is rescheduled at into the corresponding earlier time slot.
- The *Modulo_timeslot* function is called each time an actor is rescheduled to check if the required number of contiguous time slots has been made available for the target actor. In case sufficient slots are created, the *Cleanup* function terminates by returning the possible start time for the target actor determined by the *Modulo_timeslot* function. This start time may be greater than or equal to the actual ready time instance *t* of the target actor *i*.
- If not enough contiguous time slots are created after rescheduling of actors based upon their left mobility, the function begins rescheduling actors based on their right mobility. The process is equivalent to the one for left mobility based rescheduling: a *Right_mobility* function is called for computing the corresponding mobility value for each actor, actors are rescheduled in case they have non-zero right mobility, and the *Modulo_timeslot* function is called to check whether the target actor can be scheduled after another actor has been rescheduled. A difference between the two cases is that in right mobility rescheduling, the *MappedList* is sorted in a reverse priority order, i.e. an actor with lower height-based priority is considered first.
- If after neither left nor right mobility rescheduling the required number of contiguous time slots can be made available on $PE_j$, the *Cleanup* function aborts and backtracking needs to be invoked.

As mentioned above, the *Cleanup* function calls appropriate functions to determine the left and right mobility of actors. Figure 5.8 shows the flow for computing the left mobility of a given actor. The *Left_Mobility* function checks if the given actor *A* (∈ *Cleanup*'s *MappedList*) can be scheduled earlier than its current start time *tA* without violating any precedence constraints. This however also depends on the availability of the resource onto which it is mapped. The availability of a processor at a given time is determined by its corresponding entry in the MRT. The *Left_Mobility* function iteratively checks for both the conditions and tries to schedule the actor at an earlier time until one of them is violated. The overall left mobility for the actor is given by:

$$Left\_Mobility_A = Original\_start\_time_A - Final\_start\_time_A \quad (54)$$

Similarly, the right mobility of an actor *A* depends upon the slack created in the schedule by the start time of its successors. It can be computed as:

$$Right\_Mobility\_successor_A = Min[Start\_time_B] - End\_time_A, \quad (55)$$

where *B* belongs to set of successors of actor *A*. If a non-zero value is computed by the above relationship then availability of time slots on the processor is checked after the current end time of actor *A*. Assuming *slots* denotes the number of contiguous time slots available on corresponding PE after $End\_time_{A,}$ then the final value of the right mobility is the minimum of the number of available slots and $Right\_Mobility_A$ i.e.

$$Right\_Mobility_A = Min(Right\_Mobility\_successor_A, slots) \quad (56)$$

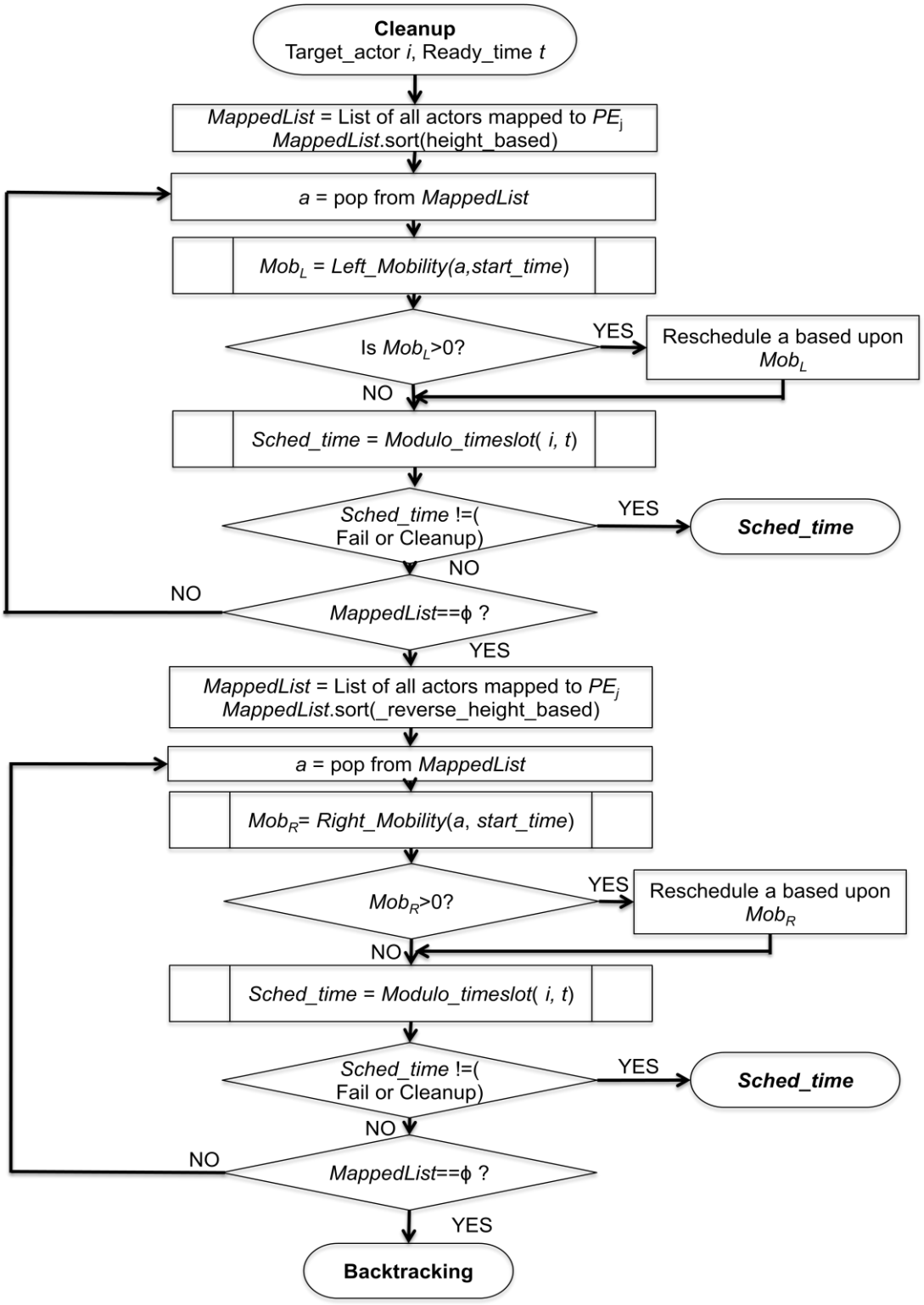


Figure 5.7: *Cleanup* function for rescheduling based on mobilities.

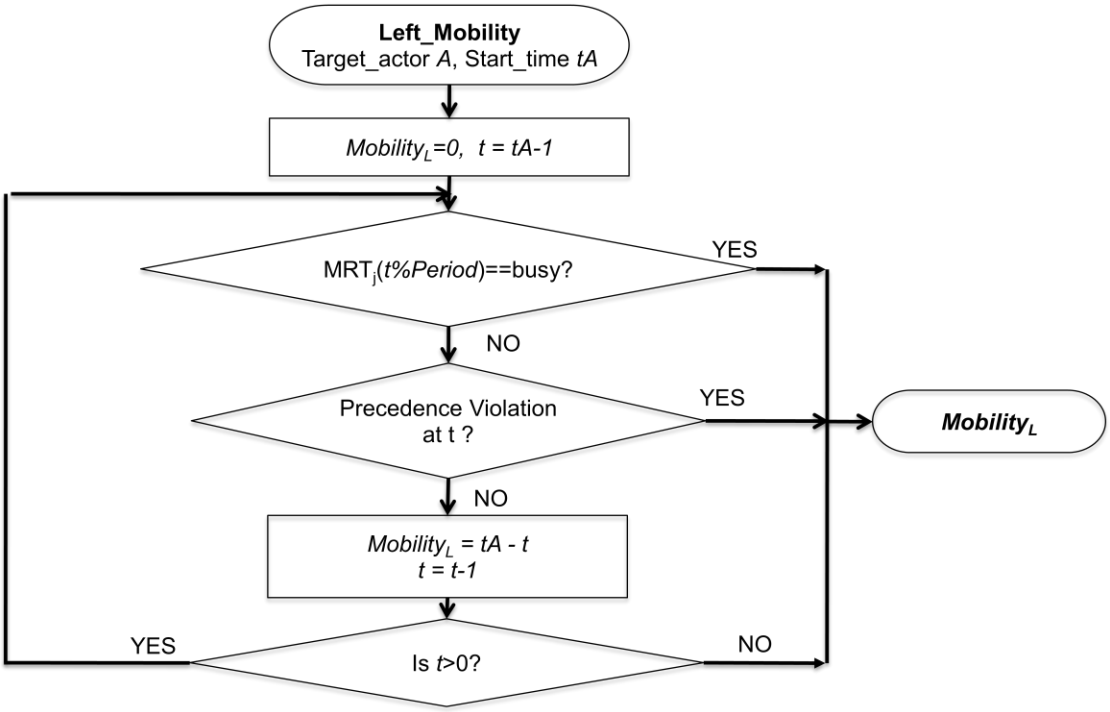


Figure 5.8: Left mobility computation.

### 5.3.3. Backtracking

As mentioned above, if the *Cleanup* function is not able to create the required number of contiguous time slots for the target ready actor even after mobility based rescheduling of the actors mapped to the target processor $PE_j$, then backtracking is required. The input to the *Backtracking* function, shown in Figure 5.9, is the target actor *i*, which is ready for execution at time instance *t* but is not scheduled due to resource non-availability. The backtracking mechanism involves two main tasks: selection of an actor that is blocking the scheduling of the target actor and unscheduling of the selected actor along with all its successors.

The key features of the function are as follows:

- Just like the *Cleanup* function, a list of all actors mapped to the target processor, *MappedList*, is generated and sorted in reverse height-based

priority. Since it is desirable to minimize the number of actors to be unscheduled, lower priority actors are considered first.

- Actors in the *MappedList* are checked whether they block the target actor and selected for backtracking accordingly. An actor *a* scheduled at time instance $Start\_time_a$ is said to be blocking target actor *i* if $Start\_time_a$ modulo *Period* is greater than or equal to *t* modulo *Period*. If it is blocking *i*, the selected actor *a* and all its successors, i.e. actors that are scheduled at a time instance greater than or equal to $End\_time_a$ (end of execution time of actor a) until the current time, are unscheduled.
- MRT and height-based priorities of the unscheduled actors are updated. The function also tracks the earliest time, *Min_time*, where unscheduling occurred.
- The *Modulo_timeslot* function is called each time an actor is unscheduled to check if the required number of contiguous time slots has been made available for the target actor. In case sufficient slots are now available, the *Backtracking* function schedules the target actor *i* at the start time determined by the *Modulo_timeslot* function. This start time may be greater than or equal to the actual ready time instance *t* of the target actor i. If backtracking was successful, *Min_time* is returned back to the main scheduler, such that it can restart scheduling from that time and hence reschedule all unscheduled actors.
- Backtracking is continued until either the target actor gets scheduled or the entire *MappedList* has been traversed without being able to create enough contiguous slots. In the latter scenario, backtracking cannot find a valid schedule and the scheduler terminates with an *infeasible* solution.

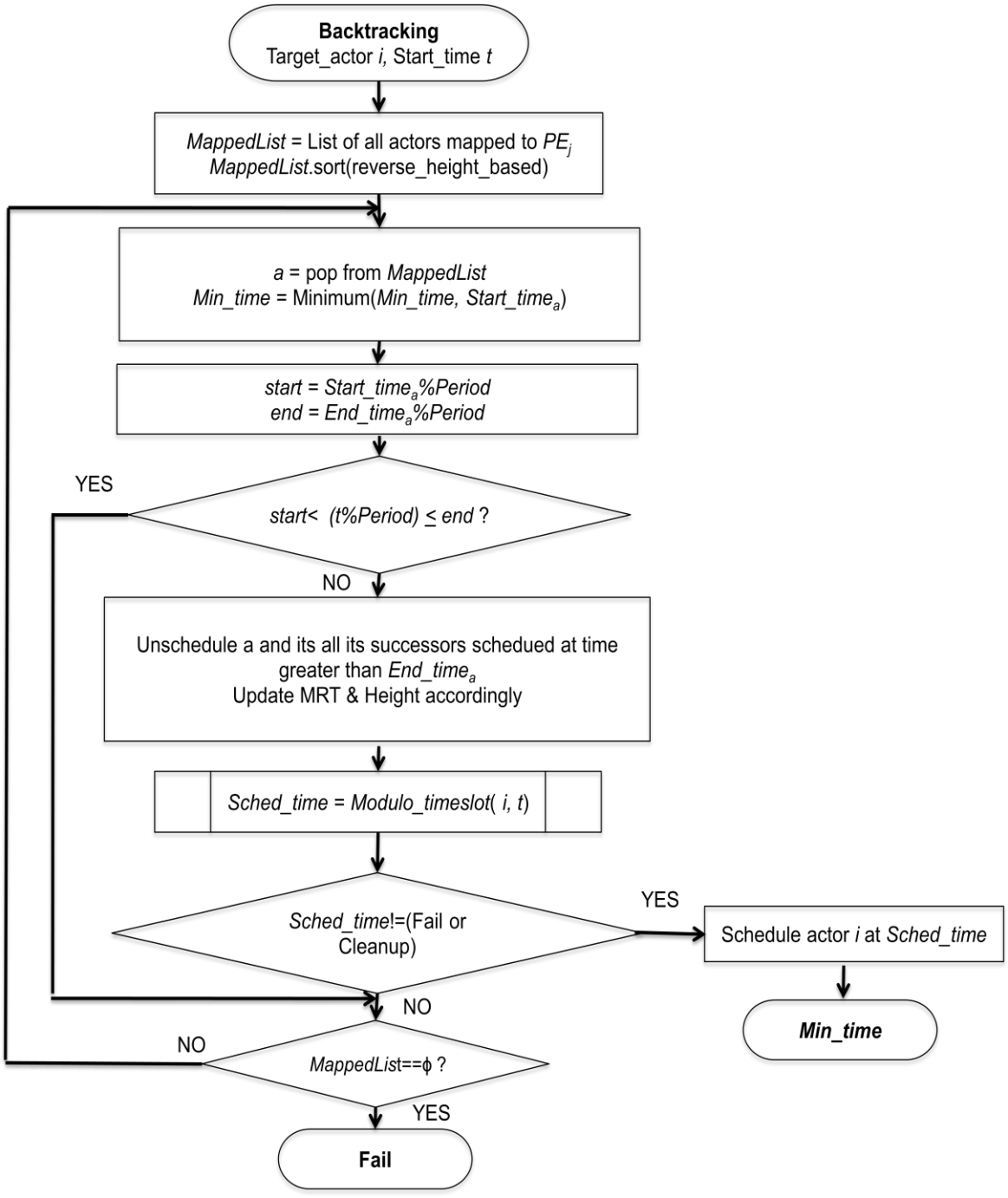


**Figure 5.9:** *Backtracking* function.

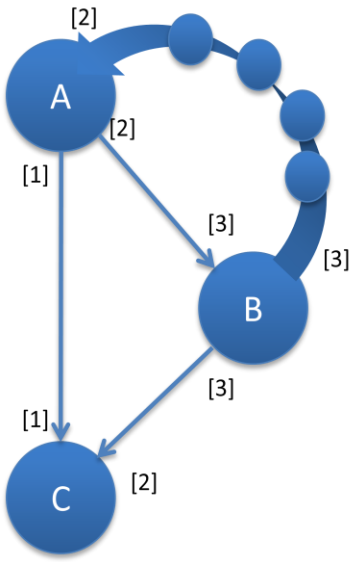


Figure 5.10: List scheduling example.

### 5.3.1. Mobility-Based Rescheduling and Backtracking Example

In this section, we explain the concept of mobility-based rescheduling and backtracking using an illustrative example. As shown in Figure 5.10, a three node consistent and live SDFG is taken as an example with the repetition vector being [3,2,3] for the node set [*A*,*B*,*C*]. A two processor architecture is considered such that actors *A* and *C* are mapped to processor $PE_1$ while actor *B* is mapped to processor $PE_2$. The execution times of the actors *A*, *B* and *C* are assumed to be 2, 2 and 3 time units, respectively. For simplicity, it is assumed that there is sufficient buffer space available on each edge at each time instance. Edge (*A*, *B*) requires 4 initial tokens. The values in square brackets represent the production and consumptions rates of the source and sink nodes at each edge. The period of the overall graph is decided by the critical processor, which is $PE_1$ in this case. The overall *Period* is 15, which is the sum of the product of execution times and repetition count of all the actors mapped onto $PE_1$.

.

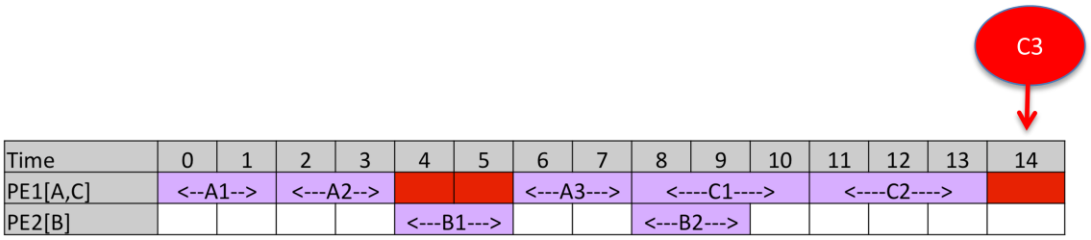


Figure 5.11: List scheduling example - step 1.

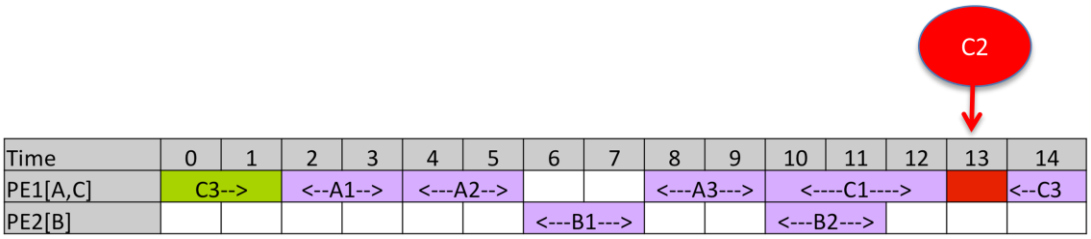


Figure 5.12: List scheduling example - step 2.

Figure 5.11 through Figure 5.15 represents the successive scheduling attempts that the list scheduler will go through to find an overlapped periodic schedule. As shown in the figures, the schedule is created for one period as a folded overlapped schedule using modulo values of a linear current time instance, For example time instance 15 will be folded back and referred to as 0 here. In the figures, different colors indicate execution of actors in different iterations. The scheduler begins scheduling with the first instance of actor *A* at time instance 0.

As shown in Figure 5.11, at time instance 14, the third instance of actor *C* (*C3)* is ready to be fired, but contiguous time slots (3 for actor *C* on $PE_1$) are not available. Three slots are available on $PE_1$, but they are not contiguous. As such, the scheduling heuristic first checks if mobility-based rescheduling is possible. At this point, there is no slack in the schedule and the scheduler resorts to backtracking. *A0* is selected as the blocking actor and unscheduled along with all its successors. Actor *C3* is scheduled starting at time 14 and the scheduler reverts back in time to begin scheduling the unscheduled actors. The schedule after unscheduling and rescheduling is shown in Figure 5.12.

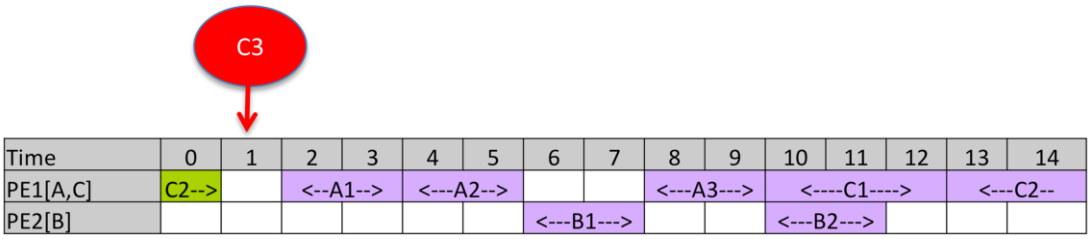


Figure 5.13: List scheduling example - step 3.

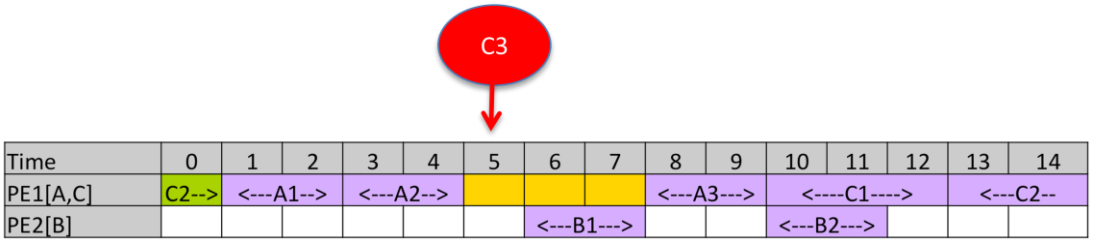


Figure 5.14: List scheduling example - step 4.

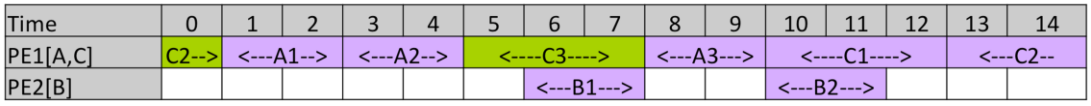


Figure 5.15: Final schedule for the example.

However, contiguous time slots now need to be created for actor *C2* starting at time 13. There is no slack and in order to schedule actor *C2*, backtracking is performed and actor *C3* is again unscheduled. As shown in Figure 5.13, contiguous time slots now need to be created for actor *C3* starting at time 1. At this point, however, actor *A1* does have a left mobility of 1. This was created because of the backtracking done in step 2. Therefore, the *Cleanup* function reschedules actor *A1* and *A2* as shown in Figure 5.14. The mobility-based rescheduling of the first and second instance of actor *A* results in creation of the required number of contiguous time slots needed for scheduling of actor *C3*. The final overlapped schedule generated by the list scheduler is shown in Figure 5.15.

### 5.3.2. List Scheduling Algorithm

The list scheduling algorithm for SDF graphs developed as part of this thesis is presented in this section. All the functions used in the algorithm are discussed in detail in the previous section. There is a pre-defined value called *budget,* which determines the maximum number of backtracking attempts that can be made by the scheduler for finding a valid schedule for a given set of inputs. Virtual source and sink are added to the dataflow graph as predecessor and successor, respectively, of all other operations (nodes) of the graph.

The algorithm is shown in Algorithm 1, wherein function *ListScheduler* is the scheduling heuristic that takes in period and mapping of the graph as input parameters from the EA. It can be divided into three main parts: prologue, kernel and epilogue. The prologue involves the initial part of the algorithm wherein static height-based priorities are computed for all the actors of the dataflow graph. It also schedules the virtual source at time instance 0, pushes it into the *UnfinishedList* and then calls the kernel. The kernel of the algorithm is the actual list scheduler, which is called repeatedly until the entire dataflow graph is scheduled or until the number of attempts at finding a valid schedule reaches the *budget* value, whichever is earlier. The key features of kernel are described in detail below:

1. Readiness of actors is checked at each time instance. All actors ready to begin their execution at the current time instance are pushed into the *ReadyList*.
2. *ReadyList* is sorted according to the height-based priorities of the ready actors so that actors farther away from the sink have a higher priority for being scheduled.
3. For each actor *x* at the top of the *ReadyList*, availability of *y* contiguous time slots, equal to the execution time of *x*, is checked on the $PE_x$ onto which *x* is mapped. Using the *Modulo_timeslot* function, the search can result in one of the following cases:
    a. *Case 1*: If contiguous time slots are available on $PE_x$, the target actor *x* is scheduled and pushed into *UnfinishedList.* In addition, the following housekeeping is performed:

i. Corresponding entries of MRT for $PE_x$ are updated
ii. All input edges of actor *x* are updated such that total number of tokens on each edge is reduced by *z*, where *z* is the number of tokens actor *x* consumes on each of its firing.
iii. Height-based priority is updated for the next instance of the given actor

b. *Case 2*: If not enough time slots are available on $PE_x$, a valid schedule is not possible for the given period of the dataflow graph. The scheduler returns *infeasible_solution* to the EA.
c. *Case 3*: If *y* time slots are available on the PE, but they are not contiguous, mobility-based rescheduling is attempted.

4. If *Modulo_timeslot* function results in Case 3, the C*leanup* function is called, which tries to create contiguous time slots on the target PE by rescheduling other actors, mapped onto $PE_x$, based on their left or right mobilities. If enough slots can be created, actor *x* is scheduled in the same way as described in Case 1.
5. If even after mobility-based rescheduling, not enough contiguous time slots are created, *Backtracking* is invoked.
6. The *Backtracking* function unschedules blocking nodes (as described in the previous section) and all its successors. This is done iteratively until either enough contiguous time slots are created for target actor *x* or until the number of backtracking attempts reaches *budget* value.
7. If enough contiguous time slots are created after backtracking, then as discussed in the previous section, the Backtracking function schedules actor *x* and returns the earliest time when unscheduling occured. The scheduler then traverses back to this time and begins scheduling the unscheduled actors.

Algorithm 5.1: *ListScheduler*(*mapping*, *period*).

```
begin
    Compute height-based priorities
    Schedule virtual source at time 0 and push it into UnfinishedList
    next_time = current_time + d_virtual_source;

    while (UnfinishedList not empty or budget>0) do
        current_time = next_time
        for all a ∈ UnfinishedList ending execution at current_time do
            Push a into FinishedList;
            Pop out a from UnfinishedList;
            Update_channel(a, finished);
        end for

        Update_readylist(FinishedList);  FinishedList = ∅

        while (ReadyList  not empty) do
            target_actor = pop actor from ReadyList;
            slots_needed = d_target_actor;
            res_m =Modulo_timeslot(target_actor,slots_needed,target_PE);
            if (res_m==Fail) then
                return infeasible solution;
            else if (res_m==Retry) then
                res_m=Cleanup(target_actor, slots_needed);
            end if
            if (res_m != Fail) then
                Schedule target_actor into time slot res_m;
                Push target_actor into UnfinishedList;
                next_time = Min(End times of actors in UnfinishedList);
            else
                res_b = Backtrack(target_actor, slots_needed);
```

```
                    if (res_b==Fail) then
                        return infeasible solution;
                    else
                        next_time = res_b;
                    end if
                end if
            end while
        end while
end
```

If a valid schedule is found, *Latency* is computed as the difference in schedule times of virtual sink and virtual source in the epilogue part of the algorithm. This value is returned back to the EA. However, if a schedule is not found, the list scheduler returns an *infeasible* solution and the EA rejects the selected mapping genome.

# Chapter 6 Experiments and Results

In this chapter, we present the results of experiments conducted to validate and evaluate the algorithms discussed in the previous chapters. A variety of randomly generated dataflow graphs and realistic examples of a sample-rate converter and an OFDM receiver have been used for this purpose. All experiments were performed on a 2.66 GHz Intel i7-920 quad core workstation with 6 GB of RAM. The SDF3 (SDF cube) [19] random graph generator has been used to create random SDFGs and CSDFGs with various attributes (execution times, graph sizes, storage sizes, etc.). The tool automatically generates graphs that are deadlock free (live) and consistent. For the experiments in this work, cyclic and acyclic, weakly connected graphs with 5 to 25 actors have been generated. The production and consumption rates, number of actors and execution times are randomly created within minimum and maximum bounds and user-specified average and variance by the tool [20].

ILP models for CSDFGs and SDFGs have been implemented using CPLEX Concert Technology for C++ [21], configured to control the optimality gap of the solution to below $10^{-6}$. The multi-objective EA has been implemented using the MOGALib Genetic Algorithm framework in C++ [22]. As proposed by [23] and following the approach in [11], the population size of the EA is taken as $1.5*(\log_2 N)$, where *N* represents the total number of permutations of chromosomes possible. In our case, *N* represents the possible mappings combinations for a given set of PEs and actors, i.e. $N = J^I$, where *J* is the total number of PEs and *I* is the total number of actors in the graph. For all experiments done, the assumed architecture has maximum number of PEs same as the number of actors in the selected graph.

In this chapter, the experimental results have been broadly categorized based on the dataflow graph type. Section 6.1 discusses the mapping and scheduling results

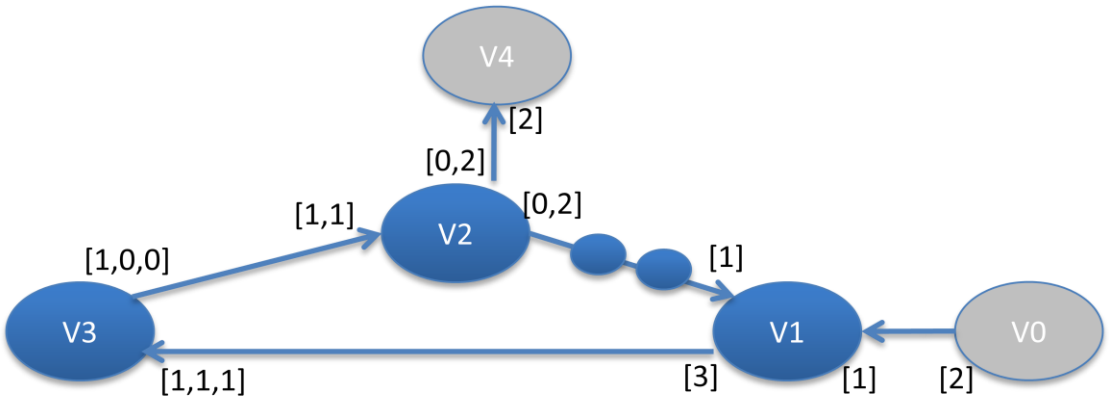


Figure 6.1: 5-Node CSDFG example.

for hybrid EA-ILP based model for CSDGs, while Section 6.2 presents the results for modified ILP and list scheduling based models as applied to SDFGs.

### 6.1. Basic ILP Model for CSDFGs

As mentioned earlier, the ILP-based scheduling solution for CSDFGs, presented in Section 5.1 is an extension of [11]. The ILP solution consists of a startup phase that precedes the stable periodic phase. An example of a CSDFG and its schedule is shown in Figure 6.1. The graph as shown consists of 5 nodes including the virtual source and sink that are added as part of the formulation. The values in square brackets represent the token consumption and production rates on the edges. Actor $V_3$ has three phases and consumes 1, 1, and 1 tokens in its first, second and third phase, respectively. Similarly, it produces 1, 0 and 0 tokens in the respective phases. The edge between actors ($V_2$,$V_1$) shows two initial tokens.

The input model assumed for the experiment is a heterogeneous architecture with three processors for which the EA generates mappings. The specifications of the EA are presented in Table 6.1. Table 6.2 shows one of the mappings produced by the EA. Processor *P0* is pipelined and has an initiation interval of 2 time units. *P0* is also the critical processor with the highest load of 8 executions of actors $V_1$ and $V_3$ combined, thus determining the minimal achievable period of 8* 2 = 16 time units. The periodic schedule generated by the ILP model for the given example is shown in

| Parameter | Value |
|---|---|
| Population size | 17 |
| Number of generations | 19 |
| Archive size | 10 |
| Crossover probability | 0.9 |
| Mutation probability | 0.1 |
| Crossover method | Uniform crossover |
| Selection method | Roulette wheel |

Table 6.1: Specifications of the EA.

| **Actor** | **$V_0$** | **$V_1$** | **$V_2$** | **$V_3$** | **$V_4$** |
|---|---|---|---|---|---|
| **Mapping** | P1 | P0 | P2 | P0 | P2 |
| **Repetition Vector** | 1 | 2 | 2 | 6 | 1 |
| **Execution Time** | 1 | 3 | 2 | 4 | 1 |
| **Period of the Graph** | 16 | | | | |

Table 6.2: Sample mapping for 5-node CSDFG example.

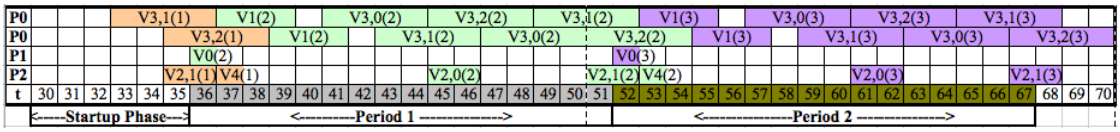


Figure 6.2: Schedule for the 5-node CSDFG example.

Figure 6.2. The different colors in the figure represent the execution of actors in different iterations. An overlapped periodic schedule is reached starting at time $t$=36.

The non-periodic startup phase ranges from $t$=0 until $t$=35 (not shown completely here due to lack of space). The optimized latency of the overall graph is 18 time units as determined by the ILP solver. Two rows have been added to show the pipelined execution of actors mapped to processor *P0*. As can be seen, the first phase of actor $V_3$ starts its execution at time instance 33 and ends it at time instance 36. However, the execution of its 2$^{nd}$ instance already begins on the PE after $II_{PE0}$ = 2 units of time have elapsed. In Figure 6.3 the Pareto-front generated for this graph is shown. To make it possible to present the results as a 3-dimensional Pareto-front, the

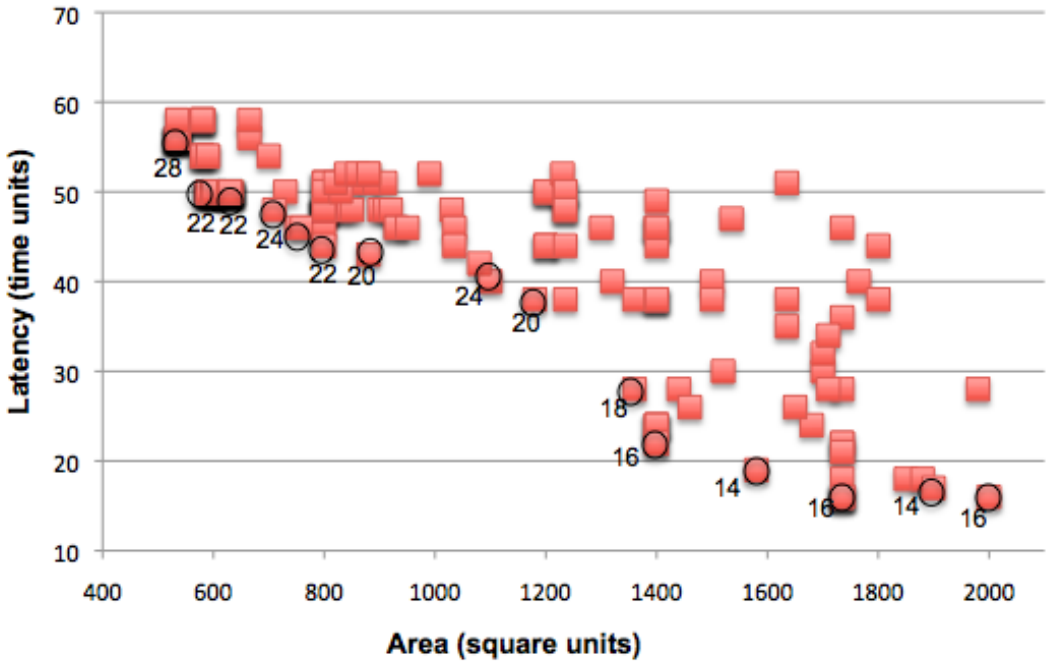


Figure 6.3: Design space for 5-node CSDFG example, Pareto-front annotated with Period values.

actual optimized buffer size (*Buffer_size*) has been folded into the total area objective (see Section 4.3), replacing the original buffer space constraint (*Buffer_space*).

The tradeoff between latency, period and area is clearly visible. The higher the area of the selected architecture, the lower are the latency and period of the overall graph. Higher area in general refers to a larger number of processing elements, hence higher possibility of exploiting parallelism in the design. Pareto-optimal design options have been encircled and annotated with their 3D Period values in the above 2D plot.

As mentioned above, in this work individual PEs are assumed to be pipelined in order to achieve an overall higher throughput for a given graph. We performed experiments to evaluate the average overhead pipelining introduces in this model. Our results show that pipelining leads to a less than 1.5% increase in run time per solution

| Architecture Type | Average runtime per solution (in seconds) | | | |
|---|---|---|---|---|
| | ILP_feasible | ILP_infeasible | EA | Total(ILP+EA) |
| Pipelined PE | 7.500 | 3.742 | 0.098 | 6.791 |
| Ujnpipelined PE | 7.595 | 3.742 | 0.106 | 6.872 |

Table 6.3: Overhead of pipelining for 5-node CSDFG.

on average. The overall run time of the scheduler is affected not only by the time per feasible solution but also by the amount of time the ILP solver spends in exhaustively searching for a solution in case of an infeasible constraint situation. As such, we also include time per infeasible ILP solution as metric in runtime comparisons. Table 6.3 shows the runtimes for a 5-node CSDFG broken down into time spent in the ILP solver per feasible or non-feasible solution and time required by the EA for generating each mapping. Overall 228 design options were considered in the run of the hybrid heuristic out of which 49 mapping options were rejected as infeasible ones by the scheduling ILP model.

Finally, we applied the ILP-based scheduler to randomly generated SDFGs and CSDFGs of different sizes. The EA was run for 20 generations with a population size of 15 for each graph i.e. a total of 300 mappings were explored for each graph. Figure 6.4 shows the exponential increase in the runtime of the ILP (averaged first over different mappings for each graph and then over 5 different random graphs of each graph size) with an increase in the graph size. A $n$-node CSDFG can always be converted to an equivalent $p$-node, $p > n$ SDFG, where additional actors are introduced to represent different phases of each original node. While the SDFG model blows up in runtime when reaching a graph size of 20 and above, the CSDFG ILP model reaches an upper limit in achievable complexity and runtime already for graph sizes of 15 and above.

## 6.2. Modified ILP and List Scheduler for SDFGs

As described in the previous chapters, besides extending the ILP-based scheduling model in [11] to CSDFGs, improvements have been made in this work to the existing SDFG formulations. The modified ILP formulation only requires reducing complexity and hence achieving lower run times. Runtime is significantly lowered further when the ILP-based solver is replaced by the list scheduler in the two-stage hybrid heuristic.

Figure 6.5 shows a runtime comparison of the three schedulers, i.e. the original ILP (Chapter 5.1, which is equivalent to [11] for plain SDFGs), the modified ILP (Chapter 5.2) and the list scheduling (LS) heuristic (Chapter 5.3). Experiments were conducted over a set of randomly generated graphs of different sizes, where 5 random graphs of each size (generated using SDF3 tool) were evaluated. The EA in all the cases was run for 20 generations with a population size of 15 each, for a total of 300 mappings explored for each graph. Each datapoint in Figure 6.5 is computed by taking an average of runtimes over the 5 random graphs for each graph size, where for each graph in the set of 5, runtime is taken as an average over all solutions corresponding to different mappings generated by the EA.

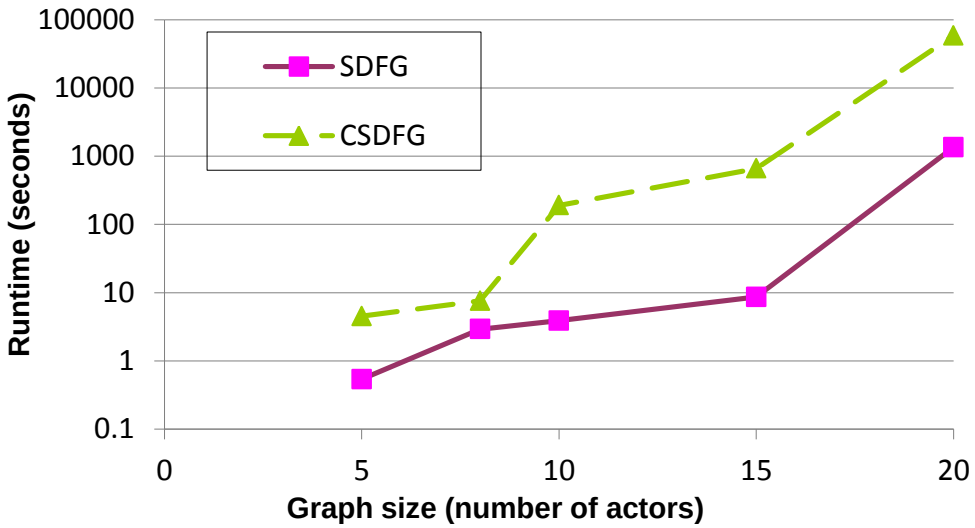


Figure 6.4: Average runtime of the CSDFG ILP model.

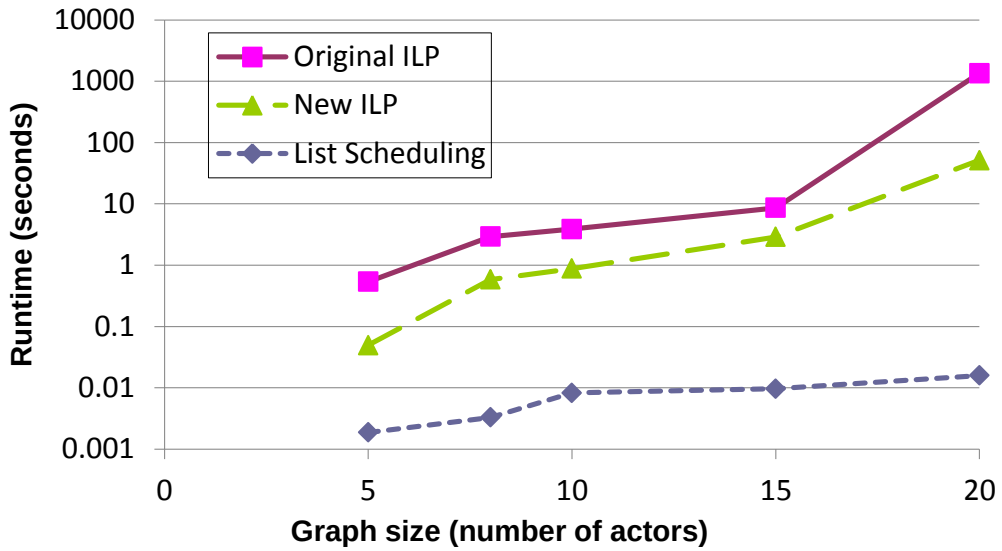


Figure 6.5: Average runtime comparison of different scheduling algorithms.

Hence, in effect each datapoint is obtained by taking an average first over different mappings for each graph and then over different graphs. Period and buffer space generated by the EA were given as input constraints to the scheduling algorithms, where no buffer size optimizations were applied in the ILPs.

As can be seen, the modified ILP achieves an average speedup of a factor of 10 in runtime when compared to the original ILP model. The runtime of ILP-based models increases significantly for graph sizes greater than 20. Therefore, experiments were limited to sizes below 20. The list scheduling heuristic, however, achieves a significant speedup in runtime as compared to both ILP-based counterparts. This speedup in runtime by a factor of almost $10^5$ of the list scheduling heuristic compared to the ILPs comes at the cost of a loss in optimality.

While ILP-based approaches are known to provide optimal solutions through exhaustive search methods, heuristics are approximations that instead aim to reduce the runtime per solution. To evaluate this optimality gap for our list scheduling heuristic, we also compared its latency results with those obtained by the ILP using the same setup as for runtime measurements (averaged over 5 graphs and 300 mappings each). Results are shown in Figure 6.6, where deviation represents the average

percentage difference in the values of latency computed by the ILP-based scheduler versus the list scheduler. As can be seen, the percentage deviation is less than 10% for graph sizes smaller than 15 actors. With an increase in graph size and hence design complexity, the optimality of the solutions obtained with the heuristic also decreases.

To compare ILP versus heuristic schedulers on a realistic example, we applied both to a sample rate converter input graph obtained from [21] (Figure 6.7). A heterogeneous processor architecture consisting of 5 PEs with randomly generated execution profiles for actors was considered. Again similar to previous experiment, buffer space and period were given as an input constraint to both ILP and list schedulers corresponding to each mapping generated by the EA. In order to achieve a fair comparison, we did not apply buffer size optimizations in the ILP. Table 6.4 presents the results from this experiment, where an optimality gap is evaluated in the results obtained from ILP versus list scheduler corresponding to the best five mappings selected in the Pareto-front generated by the EA-ILP combination.

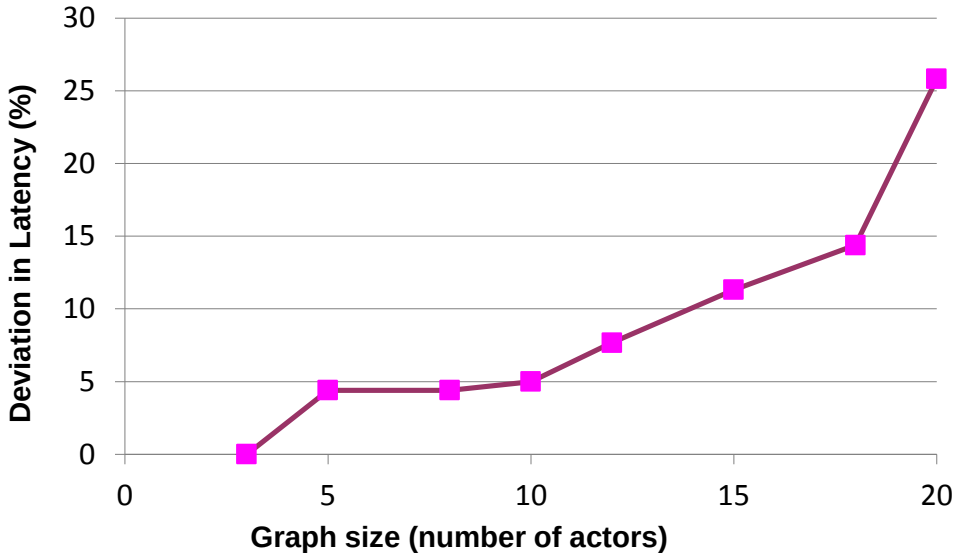


Figure 6.6: Optimality gap of Heuristic vs ILP.

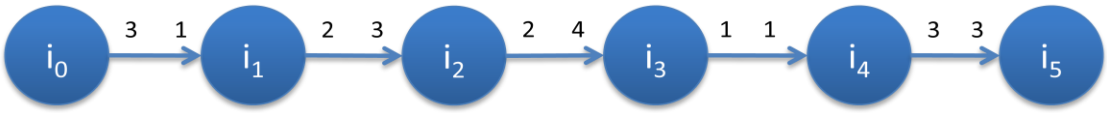


Figure 6.7: Sample rate converter.

| Period (unit cycles) | Heuristic_Latency (unit cycles) | ILP_Latency (unit cycles) | Optimality gap (unit cycles) |
|---|---|---|---|
| 14 | 52 | 48 | 4 (8.3%) |
| 17 | 44 | 44 | 0 (0.0%) |
| 18 | 44 | 44 | 0 (0.0%) |
| 20 | 44 | 42 | 2 (4.8%) |
| 23 | 37 | 35 | 2 (5.7%) |

Table 6.4: Comparison of scheduling results for sample rate converter example.

On an average over those 5 mappings, the deviation in latency was found to be below 8% over different mapping solutions. Also average runtime per solution for the ILP scheduler was 0.189s while that for the list scheduler was 0.000136s, thereby providing a significant speedup over its corresponding ILP counterpart.

As mentioned in Chapter 5, the list scheduling heuristic presented in this work is based upon an iterative modulo scheduling algorithm [7] used in the compiler domain. One main difference between the two approaches is that in this work, the concept of mobility-based rescheduling has been added to the heuristic. The original approach resorts to backtracking in cases when there are not enough contiguous time slots on a corresponding PE for a ready actor to execute. By contrast, in mobility-based rescheduling, an attempt to reschedule other actors mapped to the same PE based on their free mobility is first made. Only if rescheduling does not create enough contiguous slots, backtracking is applied. To evaluate the effectiveness of mobility-based rescheduling, we applied list schedulers both with and without rescheduling to a set of randomly generated SDFGs of different sizes. Table 6.5 shows a comparison of the number of backtracking attempts made and the latency values obtained for the two

| Graph Size | | Only Backtracking | | Mobility based rescheduling + backtracking | |
|---|---|---|---|---|---|
| | Period | #Backtracks | Latency | #Backtracks | Latency |
| 3 | 15 | Max | N/A | 2 | 22 |
| 5 | 16 | 0 | 16 | 0 | 16 |
| | 10 | 0 | 16 | 0 | 16 |
| | 11 | 3 | 23 | 3 | 23 |
| | 12 | 0 | 24 | 0 | 24 |
| | 13 | 0 | 21 | 0 | 21 |
| | 14 | 2 | 28 | 0 | 28 |
| | 17 | 2 | 18 | 1 | 18 |
| 10 | 25 | Max | N/A | 0 | 63 |
| | 41 | Max | N/A | Max | N/A |
| | 34 | Max | N/A | Max | N/A |
| | 22 | 4 | 70 | 3 | 70 |
| | 19 | 4 | 51 | 3 | 51 |
| | 20 | 4 | 38 | 3 | 38 |

Table 6.5: Mobility-based rescheduling vs. unmodified backtracking.

approaches. The maximum number of backtracks allowed in the algorithm (i.e. the *budget* value) was set to 10. As can be seen in the Table 6.5, in the given set of graphs there were 2 cases where our heuristic was able to find a valid solution whereas a method using only backtracking failed to do so. Also, it is evident that if mobility-based rescheduling is performed, the number of required backtrackings and hence overall complexity and runtime is less. There were however a couple of scenarios where the neither of the list scheduling based heuristics could yield a valid schedule. Those cases have been shown as N/A in the table.

The overall two-stage heuristic with an inner list scheduling kernel embedded into the global EA framework was further characterized and validated on industrial-strength design example: an OFDM receiver. Figure 6.8 shows the design space and Pareto-front generated by the EA-LS based scheduling heuristic. The OFDM receiver is a highly sequential design and does not provide opportunities for resource sharing. Therefore, the mapping of actors to the architecture was fixed and hence, the overall latency of the graph and the cost of the PEs (resources) stayed constant. In order to

explore possible options for this design, the EA was modified to take different period values as additional exploration dimension instead of generating mappings (which were fixed for the given example). Generated period values were explored in a range starting from the minimum period of the critical processor (30000 time units in this case) along with different buffer sizes on the edges of the graph. Hence, in this experiment, period was taken as a decision variable instead of an objective function. As shown in Figure 6.8, variations in throughput of the design could be explored by varying the buffer sizes on the edges of the graph. The squared data points in the plot represent the design options that lie on the Pareto-front. As expected, increasing the buffer spaces on the edges directly impacted the period of the overall graph, i.e. as the buffer space available on edges approaches their maximum values, the period of the overall graph reduces, thereby increasing the throughput of the overall design. For this example, overall 150 design options were considered, which took 22.43 seconds for the entire run. The list scheduler took 7.988 seconds on an average per solution and the overall average time per mapping and scheduling solution was 8.882 seconds (GA+LS).

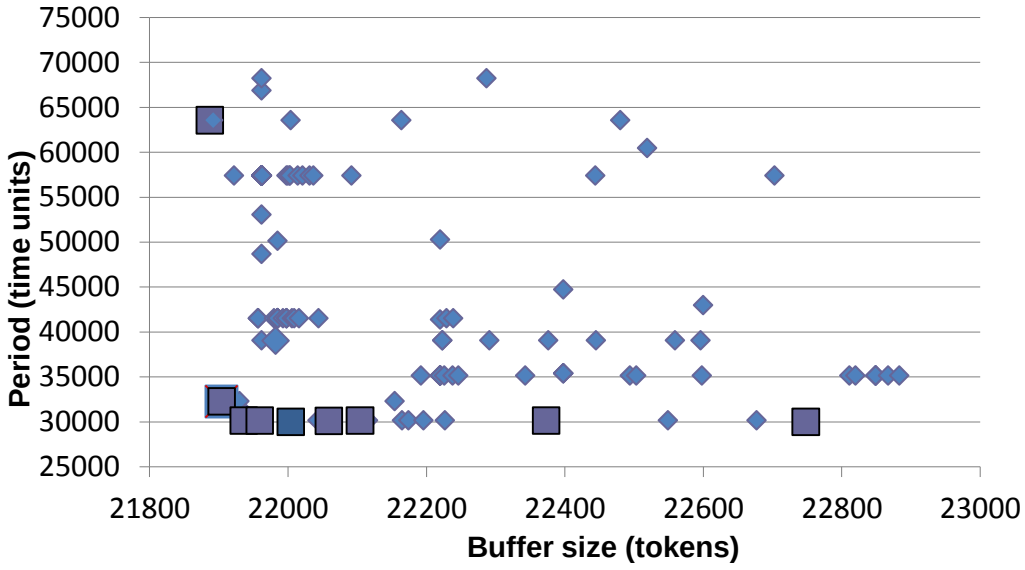


Figure 6.8: Design space for OFDM receiver.

# Chapter 7 Summary and Conclusion

The aim of this thesis work was to build upon, explore and improve existing mapping and scheduling heuristics for dataflow graphs. In this work, the multi-objective mapping-scheduling optimization presented in Jing et al. [11] has not only been extended from SDFGs to CSDFGs, but also several scheduling optimizations have been proposed both in the ILP and in the heuristic domain. A modified ILP formulation presented in this work achieves an average speedup in runtime by almost a factor of 10 while maintaining the same optimality in the solutions as the ILP presented in Jing et al. [11]. In addition, buffer optimizations have been added in this work. Both shared memory and distributed memory models were explored and implemented. A further advantage in the overall throughput of the graph is obtained by supporting pipelined processing elements.

In addition to an improved ILP, a novel list scheduling heuristic based on an adapted iterative modulo scheduling algorithm has been developed. This heuristic has been successfully evaluated on multiple sets of randomly generated graphs and realistic design examples. It provides a significant speedup in runtimes while maintaining near-optimality of the solutions within an acceptable gap of 10% when compared to its ILP counterparts. The list scheduling heuristic presented in this work has been extended by introducing a new concept of mobility-based rescheduling before resorting to backtracking. Results have shown that this increases the probability of finding a legal and valid schedule, thus increasing the optimality of the list scheduling approach.

In summary, this work contributes towards efficient design space exploration of real-time streaming signal processing designs on heterogeneous target architectures. Presented solutions can employed at any design stage for evaluating the optimality and feasibility of the selected design option. ILP-based solutions were initially considered for establishing a baseline of achievable optimality of solutions. The final list scheduling heuristic has been successfully validated on several practical, industry-

strength design examples. In future work, the presented approach can be further extended to more complex dataflow graphs, such as parameterized or scenario-aware dataflow models. Furthermore, additional heuristics can be explored as scheduling algorithms to further optimize runtime efficiency and optimality.

# References


[1] S. Stujik, M. Geilen and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proceedings of the Design Automation Conference (DAC)*, 2006.

[2] S. Stujik, M. Geilen and T. Basten, "Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs," in *IEEE Transactions on Computers*, Volume 57, Issue 10, pp. 1331-1345, Oct 2008.

[3] R. Govindrajan, G. Gao, "Rate-optimal schedule for multi-rate DSP computations," in *Journal of VLSI Signal Processing*, Volume 9, pp. 211-235, 1995.

[4] A. Stoutchinin, "An integer linear programming model of software pipelining for the MIPS R8000 processor", in *Parallel Computing Technologies*, 4th International Conference, 1997.

[5] S. Groot, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Journal*, 1992, log number 9108174.

[6] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW machines," *in Proceedings of the ACIM SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[7] B. Rao, "Iterative Modulo Scheduling," in *The International Journal of Parallel Processing*, Volume 24, 1996.

[8] G. Goossens, J. Rabaey, J. Vandewalle and H.D. Man, "An efficient microcode compiler for application specific DSP processors," in *IEEE Transaction on Computer-Aided Design*, Volume 9, Number 9, pp. 925-937, Sept 1990.

[9] D. Gajski, S. Abdi, A. Gerstlauer and G. Schirner, "Embedded System Design: Modeling, Synthesis and Verification", ISBN 978-1-4419-0503-1, Springer, 2009.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow graphs for digital signal processing," in *IEEE Transaction on Computers*, Volume 36, Number 1, pp. 24-35, 1987.

[11] J. Lin, A. Srivastsa, A. Gerstlauer and B. Evans, "Heterogeneous multiprocessor mapping for real-time streaming systems," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASS)*, 2011.

[12] J. Zhu, I. Sander and A. Jantsch, "Buffer minimization of real-time streaming application scheduling on hybrid CPU/FPGA architectures," in *Proceedings of the IEEE Conference on Design Automation and Test in Europe (DATE*, 2009.

[13] R. Doemer and W. Chen, "A Fast Heuristic Scheduling Algorithm for Periodic ConcurrenC Models," *Proceedings of the IEEE Design Automation Conference (ASP-DAC)*, 2010.

[14] Y. S. Chiu, C. S. Shih and S. H. Hung, "Pipeline schedule synthesis for real time streaming tasks with inter/intra instance precedence constraints," in

*Proceedings of the IEEE Conference on Design Automation and Test in Europe (DATE)*, 2011.

[15] W. Kim, J. Y. Chang and H. Cho, "Pipelined scheduling of functional HW/SW modules for platform based SOC design," in *ETRI Journal,* Volume 27, Oct 2005.

[16] T. Shin, O. Hyunok and H Soonhoi, "Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph," in *Proceedings of IEEE Design Automation Conference (ASP-DAC),* pp. 165-170, 2011.

[17] S. Bhattacharya, P. Murthy and E. Lee, *Software synthesis for dataflow graphs*, Springer, 1996.

[18] J. Lin, A. Gerstlauer and B. Evans, "Communication-aware heterogeneous multiprocessor mapping for real-time streaming systems", in *Journal on Signal Process Systems,* Springer, May 2012.

[19] T. Parks, J. Pino and E. Lee, "A comparison of synchronous and cyclo-static dataflow", in *Proceedings of the IEEE Conference on Signals, Systems and Computers,* 1995.

[20] S. Stuijik, M. Geilen and T. Basten, "SDF3: SDF for free," in *Proceedings of the IEEE Conference on Application of Concurrency to System Design*, pp. 276-278, 2006. Available at http://www.es.ele.tue.nl/sdf3.

[21] C. Optimization Using the CPLEX callable library and CPLX mixed integer library, CPLEX Optimization, Incline Village, 1993.

[22] R. E. Rosenthal, “GAMS- A User’s Guide,” GAMS Development Corporation, Washington, DC, USA, 2012.

[23] J. T. Alander, “On Optimal population size of genetic algorithms,” in *Proceedings of the IEEE Conference on Computer Systems and Software Engineering*, pp. 65-70, 2002.