Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Jakub Kopřiva**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **Algorithms for Mapping and Scheduling Real-Time Streaming Applications on Heterogeneous Multi-core Platforms**

Guidelines:

This assignment addresses mapping of streaming applications on multi-core systems. It deals with design of algorithms for automated mapping and scheduling. Then on the basis of the existing algorithms and literature research, a mapping and scheduling algorithm will be designed. The expected outcome of the assignment is an implementation of the proposed methodology.

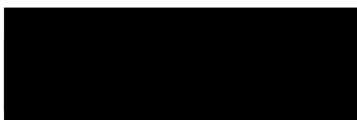
- Specify the mapping and scheduling problem and review existing solutions.
- Study the designed algorithm and justify choices made during the design phase.
- Describe possible design adaptations considering the specific problem.
- Implement selected/designed algorithm(s) using ILP CPLEX solver.
- Evaluate the quality of the proposed algorithm in terms of time and space complexity.
- Compare the designed algorithm with existing algorithms using existing models of real applications from the SDF3 tool [2]. The ILP-based approach in [1, 3] will serve as baseline for comparison.

Bibliography/Sources:

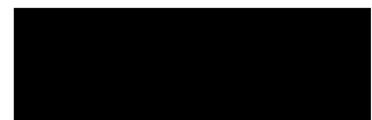
[1] Jing Lin; Srivatsa, A.; Gerstlauer, A.; Evans, B.L., "Heterogeneous multiprocessor mapping for real-time streaming systems," Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on , pp.1605,1608, 22-27 May, 2011
[2] Stuijk, Sander, Marc Geilen, and Twan Basten. "SDF3: SDF For Free. Sixth International Conference on Application of Concurrency to System Design, 2006.
[3] J. Lin, A. Gerstlauer, and B. L. Evans. "Communication-aware heterogeneous multiprocessor mapping for real-time steaming systems."; Journal of Signal Processing Systems Vol. 69., no. 3, 2012.

Diploma Thesis Supervisor: Dr. Benny Akesson, MSc.

Valid until the end of the summer semester of academic year 2015/2016



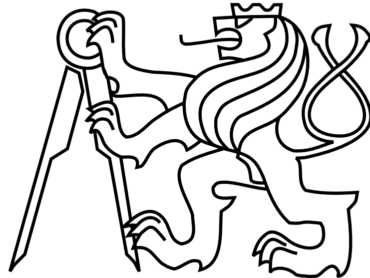
prof. Ing. Jiří Žára, CSc.
Head of Department



prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 24, 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Algorithms for Mapping and Scheduling Real-Time Streaming Applications on Heterogeneous Multi-core Platforms

Bc. Jakub Kopřiva

Supervisor: Dr. Benny Akesson, MSc.

Study Programme: Open Informatics

Field of Study: Software Engineering

May 10, 2015

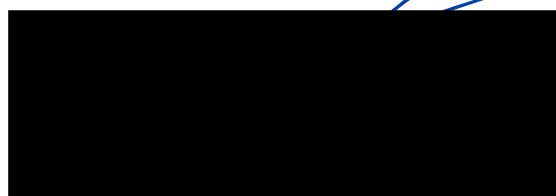
Aknowledgements

I would like to thank both my consultant, Ing. Přemysl Šůcha, Ph.D and my supervisor, MSc. Benny Akesson. They were inspiring me and leading me through the master thesis. Also, I can not forget to thank all OF my family for supporting me during the studies.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague on May 11, 2015



Abstract

This thesis addresses the problem of mapping and scheduling of real-time data driven applications on heterogenous multi core platforms. We are proposing an ILP formulation to solve this problem and introduce multiple performance upgrades for the formulation in usage of lazy constraints and a symmetry breaking algorithm. Also, we describe the pros and cons of two options and how to use lazy constraints. We provide an implementation tutorial for lazy constraints with use of CPLEX Java API. We are providing experiments for lazy constraints and the symmetry breaking algorithm as well as for baseline ILP for comparison. We experimentally show that the performance upgrade is up to factor 7, but based on results of real applications, we are still proposing to work on the performance issues in the future.

Keywords: mapping, scheduling, real-time application, heterogenous multi core platform, lazy constraints, symmetry breaking, ILP, CPLEX Java API

Abstrakt

Tato diplomová práce se zabývá problematikou mapování a rozvrhování aplikací prováděných v reálném čase na heterogení multiprocesorové platformy. Pro vyřešení problému je v práci navržena formulace celočíselného programování. Součástí práce je také několik vylepšení daného algoritmu za účelem zvýšení jeho výkonu. Tyto přístupy jsou porovnány na základě experimentů z hlediska výkonnu jednotlivých algoritmů. Práce poskytuje stručný návod jak používat lazy constraints pomocí CPLEX Java API a experimentálně ukazuje, že je možné až sedmkrát zrychlit navržený algoritmus. Na základě experimentů s reálnými aplikacemi navrhuji dále pracovat na problémech spojených s výkonnem.

Klíčová slova: mapování, rozvrhování, aplikace prováděné v reálném čase, heterogení multiprocesorové platformy, lazy constraints, odstraňování symetrií, celočíselné programování, CPLEX Java API

Contents

1 Introduction **1**

2 Background **3**
2.1 Synchronous Dataflow Model 3
2.2 Homogeneous SDF Model 5
2.3 System Architecture 6
2.4 Mapping and Scheduling relation 6

3 Problem Formulation **9**

4 Baseline ILP **11**
4.1 ILP Formulation 11
4.2 Baseline ILP improvement 15
4.3 Linearization of the baseline ILP 15

5 Lazy constraints **17**
5.1 Introduction to lazy constraints 17
5.1.1 Lazy constraints function 17
5.1.2 Lazy constraints callback 19
5.2 ILP with lazy constraints 20
5.3 Symmetry breaking 22
5.3.1 Algorithm to find symmetries 23
5.3.2 Symmetry breaking using lazy constraints callback 25
5.4 CPLEX parameters affecting performance 29

6 Experimental Setup **31**
6.1 Benchmark instances 31
6.2 Hardware 34
6.3 Baseline experiments using IBM OPL Studio 34
6.4 Baseline ILP Java experiments 34
6.5 Baseline ILP with lazy constraints 35
6.6 Symmetry breaking experiments 36

7 Related works **41**

8 Conclusion and Future work **43**

Bibliography **45**

A Graphs used for benchmarking **47**

B Nomenclature **65**

C Content of attached CD **67**

List of Figures

2.1 Simple graph 3
2.2 Example of instance which has finite memory buffer between actors . 4
2.3 Graph containing self edge at actors a_1 and a_2 4
2.4 Graph shows how is possible model finite buffer from figure 2.2 5
2.5 Schedule showing transient and periodic phase 6
2.6 Multiprocessor system 7

4.1 SDF graph with indexes introduced in equation (4.2) 12
4.2 Schedule showing variables used by equation (4.3) 13

5.1 Symetrical schedules 23
5.2 Matrix of WCET with symmetries 23
5.3 Symmetrical mapping 23
5.4 Symmetries for matrix introduced on figure 5.2 25
5.5 Initial mapping and symmetries for the instance 26

6.1 H263 Decoder with buffer 32
6.2 Modem with buffer 33
6.3 Input of partially heterogenous instance instance of MP3 decoder . . 35
6.4 Input of fully heterogenous instance instance of MP3 decoder 36
6.5 Solving time of first instance set in real time 37
6.6 Solving time of second instance set in real time 38
6.7 Solving time of second instance set in CPU time 38

A.1 Satellite receiver with buffer 48
A.2 Sample-rate converter with buffer 49
A.3 MP3 decoder with buffer 50
A.4 Modem with buffer 51
A.5 H.263 decoder with buffer 52
A.6 Artifical instance 1 53
A.7 Artifical instance 2 54
A.8 Artifical instance 3 55
A.9 Artifical instance 4 56
A.10 Artifical instance 5 57
A.11 Artifical instance 6 58
A.12 Artifical instance 7 59
A.13 Artifical instance 8 60

A.14 Artifical instance 9 61
A.15 Artifical instance 10 62
A.16 Artifical instance 11 63
A.17 Artifical instance 12 64

C.1 Content of attached CD 67

List of Tables

4.1 Input variables 12
4.2 Decision variables 13

6.1 Size of the real application instances used for benchmarking 32
6.2 Specification of hardware used for experiments 34
6.3 Results of experiments using Kepler server for calculation 35
6.4 Results of the experiments using lazy constraints on MacBook Pro . . 39

Chapter 1

Introduction

With an increasing number of real-time streaming applications, we must also keep in mind the importance of satisfying real-time constrains. Besides non-real-time applications, which do not have any requirements, there are two types of real-time constraints. First, there are soft real-time constraints with requirements to avoid missing deadline, but some misses are tolerable. The second type is hard real-time constraints which do not tolerate misses of any deadline. Missing deadlines on hard real-time constraint applications would have catastrophic consequences. Normally, applications have requirements for execution time, but streaming applications care about throughput, which stands for the amount of data produced by the application per unit of time. For example, a video decoder would usually have a minimal throughput constraint of 30 or 60 frames per second (FPS). To satisfy throughput, we need to consider the hard real-time constraints.

Under best case violation of these constrains, it will get you a lower frame rate on your TV but real-time applications started to affect our life even more than ever before. These applications are used in the automotive industry for keeping a car between driving lines and other safety features. While the car still has breaks, aircrafts do not while in the air, a violation of the real-time constraints could have catastrophic consequences. We also need to consider the military usage of real-time streaming applications e.g. in unmanned aerial vehicle (UAV) drones, where camera feeds are transported across huge distances through satellite connections. This means there is a limited bandwidth and a necessity to use a video encoder and decoder. These constraints are largely related to the application

The problem in this thesis is to synthesize an optimal mapping of application tasks to processor cores and derive a static execution schedule. We are considering real-time streaming applications on a heterogeneous multiprocessor system-on-chip (MPSoC) platform while making sure that we satisfy the real time constraints. For this purpose, we are using an integer linear programming (ILP) formulation. In terms of finding an optimal schedule, we also need to take into account the mapping decisions for the processors and communication channels because it has a big impact on the schedule. Heterogeneous platforms have become very popular for their high computational performance at low power in the past few years. To evaluate our approach, we used real application benchmarks provided by the SDF^3 tool [13] and

also propose several artificial benchmark instances. Mapping and scheduling should not take too long as to avoid impacting the design time of application.

Our goal was to synthesize an optimal mapping and schedule for the application model in the form of synchronous dataflow (SDF) graph with the goal of maximizing throughput. It hold a set of tasks, precedence constraints and a worst-case execution time for each processor where the actor is able to execute.

This thesis consists of multiple parts. Chapter 2 provides the background information required to understand the contributions of the thesis. In Chapter 3, we formulate the problem for this thesis. We present an ILP formulation which will serve as the baseline formulation for comparison in Chapter 4.

Use of this ILP formulation has proven to be ineffective in terms of performance (solving time) on real application models. Hence, we are proposing improvement for this ILP in Chapter 4 while also using lazy constraints. A description and usage of lazy constraints with a symmetry breaking algorithm can be found in Chapter 5.

In Chapter 6, we introduce an experimental setup with benchmarks which came from the proposed algorithms. This chapter contains tables and charts that can easily be compared to the baseline ILP. We are introducing related works which has been done in the field of mapping and scheduling for real-time streaming applications in Chapter 7. The conclusion is described in Chapter 8 where we summarize what has been achieved in the thesis as well as future work suggestions.

Chapter 2

Background

In this section we discuss the application model used as an input to our algorithms, system architecture and the close relation between mapping and scheduling.

The application must satisfy real-time requirements, and between those requirements, they must satisfy the throughput. Throughput is basically a rate in which real-time streaming applications can produce data for the output. Throughput needs to be satisfied e.g. in order to produce a sufficient frame rate for a video streaming application.

2.1 Synchronous Dataflow Model

Synchronous dataflow models are used to describe real-time applications. A SDF model can be easily represented by a graph. SDF graphs consists of nodes called actors (a_i) and directed edges called channels which represent the connections between actors where the data is transported from one actor to another. The data is usually represented as tokens (a batch of data with an exact size). Tokens are transported through the channel in FIFO order.

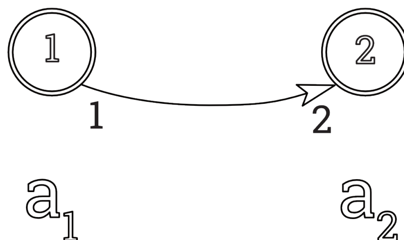


Figure 2.1: Simple graph

Each node is denoted by a worst-case execution time (WCET) which is usually placed inside the node. WCET represents the execution time in the worst case scenario i.e. actors cannot execute slower than WCET. The channel is the connection between two actors. A channel starts in source actor a_s and is directed to target actor a_t. A channel is denoted as production P_{a_s,a_t} and consumption rate C_{a_s,a_t} next to the source

and target actors respectively. We should also introduce terms of firing, which means firstly the actor atomically consumes a number of tokens from each incoming channel (the number of tokens consumed is equal to the consumption rate on the particular incoming channel). After, the actor executes for its WCET and atomically produces a number of tokens which are equal to the production rate of each outgoing channel.

For example, in order to fire actor a_2 from figure 2.1, we need to fire the actor a_1 at least twice. This would ensure we have enough tokens for actor a_2 for it to be ready to fire.

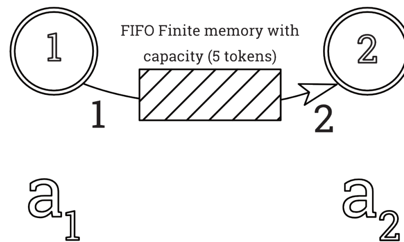


Figure 2.2: Example of instance which has finite memory buffer between actors

A self edge, with one initial token from actor a_2 to a_2 ensures the actor will only fire once in a given moment, can be seen in figure 2.3. The initial token is the amount of data present on the channel even before the application starts executing. Initial token is shown as dot on the edge in a graph.

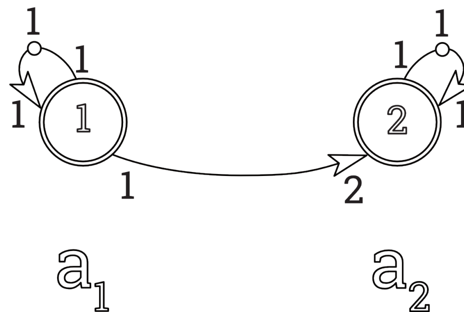


Figure 2.3: Graph containing self edge at actors a_1 and a_2

The self edge works in a way where the actor a_1 takes 1 token (the initial token) to start firing. While it is executing, it cannot start executing again because there are not enough tokens at each input channel. Once the execution stops, the actor produces another token on the self edge and is ready to fire again. The actor a_1 also produces one token through channel $a_1 \longrightarrow a_2$. To fire a_2, actor a_1 needs to fire at least twice as seen from figure 2.3

Initial tokens can also be combined with a back edge to model a finite buffer between actors. Figure 2.4 displays how it is possible to model a finite FIFO buffer introduced in figure 2.2. The application starts with initial tokens on the back edge $a_2 \longrightarrow a_1$. Right before a_1 starts firing, it will consume one token from the back edge which means it is reserving space in the FIFO memory (this ensures there is space

for the output to be placed in the buffer). Once actor a_2 starts firing, it will consume tokens on the channel $a_1 \longrightarrow a_2$. After the firing has finished, it produces the same amount of tokens it consumed for the firing on the back edge $a_2 \longrightarrow a_1$ which means it frees the buffer memory for the amount of data used for firing.

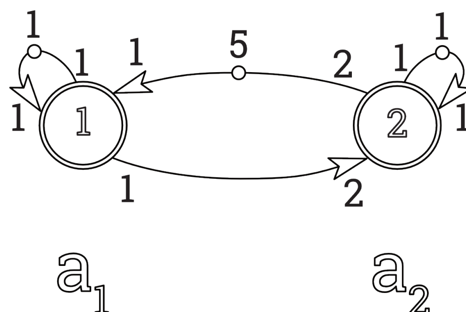


Figure 2.4: Graph shows how is possible model finite buffer from figure 2.2

Schedule of application from the graph in figure 2.4 can be viewed in figure 2.5. Every schedule for real-time application consists of a transient phase and a periodic phase. The schedule for a transient phase can vary from the schedule of periodic phase because during the transient phase, the application is preparing enough input data (tokens) on the channels for periodic execution. Each time the periodic phase executes, it generates an output for the entire application.

In this thesis, we use the term of repetition vector. It consist of the number of firings needed in the periodic phase for each actor. The repetition vector for our example application in figure 2.4 would be $n = \{2, 1\}$.

2.2 Homogeneous SDF Model

A homogeneous synchronous dataflow (HSDF) model is a special type of SDF model where every actor is executed only once during a period. This means a repetition vector of HSDF model holds only items equal to 1. The repetition vector is discussed further in section 2.4

HSDF and SDF models of the same application is mutually convertible by an algorithm, although the complexity of those algorithms can be very high. Depending on the algorithm used and the graph complexity, we can see the exponential increase of graph size in terms of the number of nodes and edges, it is described more in [1]. The article is also provides an algorithm which lowers the graph complexity of the SDF - HSDF conversion. This algorithm is part of the SDF^3 tool [13]. Several algorithms implemented in the SDF^3 tool were also used to prepare the data for our experiments.

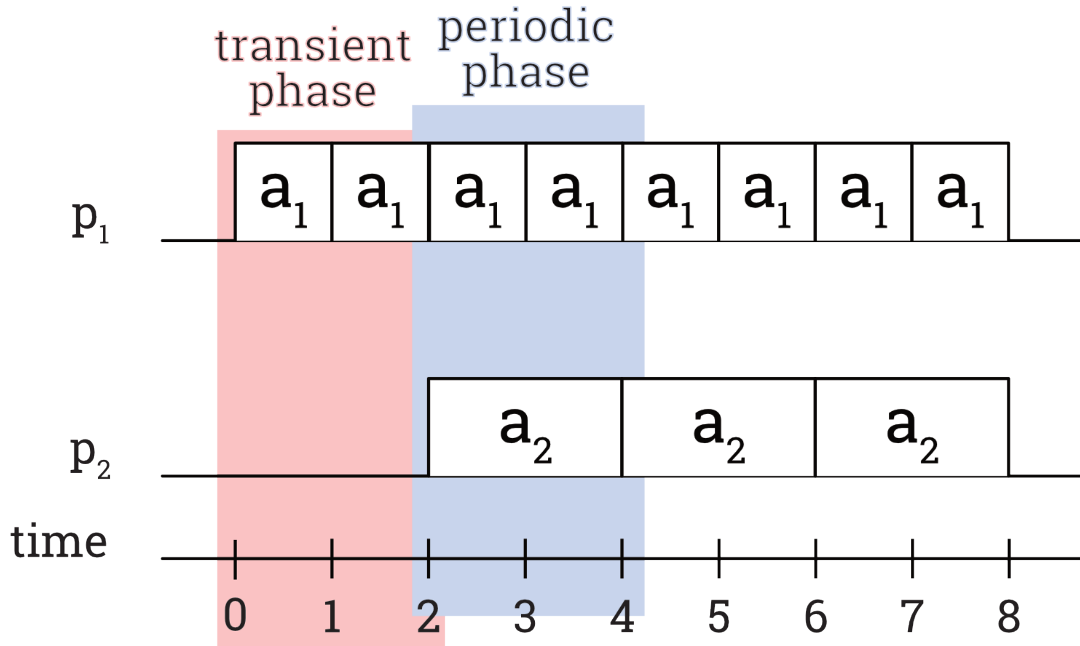


Figure 2.5: Schedule showing transient and periodic phase

2.3 System Architecture

The architecture of the system has been considered in several ways. Firstly, we need to point out that all the considered architectures were multiprocessor multicore systems. Communication between the cores and processors have not been considered due to their calculation complexities.

We assume m cores of k different types. If $m = k$, we consider it as a fully heterogenous system. This means that all actors have a different WCET for each processor/core. If $m > k$, then the system architecture is only partially heterogeneous which means all actors have the same WCET for the same type of processor e.g. graphics processing unit (GPU) .

2.4 Mapping and Scheduling relation

Mapping refers to assigning actors to processors. This means each actor will have a dedicated processor during the application runtime while the static schedule will ensure the processor will be available for concrete actor execution at a particular time.

Scheduling means building a static schedule for every single actor in the application. During this, we optimize particular criteria. For example, the overall length of schedule or just part of the schedule e.g. periodic phase.

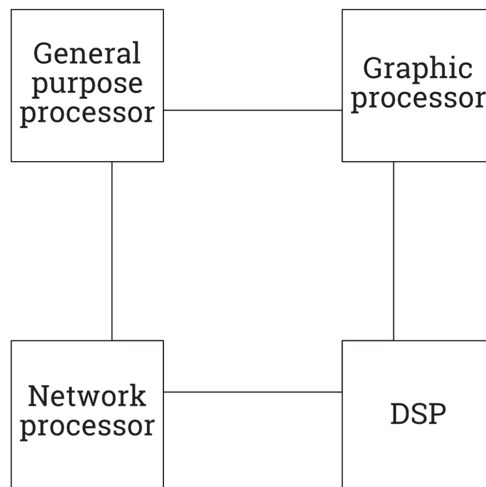


Figure 2.6: Multiprocessor system

Mapping and scheduling are strongly connected one to another. There can be a big difference in schedule if an actor is mapped to a different processor due to the possible difference in WCET or by overwhelming the processor by actors while other processors are free.

A schedule consists of a transient and a periodic phase as displayed in figure 2.5. While the transient phase is executed only once after the application has started, the periodic phase is periodically executed to generate the output. Inversion of the length of periodic phase $\left(\frac{1}{Period}\right)$ is called throughput and it represents the frequency the output generates by the application. To generate an output, an application usually needs to run through multiple states. Each application state is also represented in the graph, where channels contain different numbers of tokens in each state.

A period is the amount of time in which the application runs and data is generated on output of the application. After the beginning of the periodic phase, the application returns to the initial state of the period in terms that channels are holding the exact same amount of tokens and actors are in the same phase of execution (period can start during firing of actor).

Repetition vector has been briefly introduced in section 2.1. A vector consists of the number of executions needed for each actor during one period. A repetition vector expresses how many times an actor needs to fire to ensure the application will generate data for the applications output.

Chapter 3

Problem Formulation

We have been given a SDF model to work with. The model was briefly described in section 2.1. We also assume that analysis in terms of calculating the correct repetition vector was performed. The repetition vector is also briefly described in section 2.1 and 2.4. With the SDF model given, the matrix d consists of WCET for actors per possible mapping decision $d_{i,j} = WCET_{i,j}$. Matrix d relates to considered architectures. If $d_{i,j-1} \neq d_{i,j}\ \forall i \in I\ \forall j \in J \backslash \{1\}$, $I = \{1, 2, \ldots, n\}$ and $J = \{1, 2, \ldots, m\}$ where n stands for total number of actors and m stands for number of resources (processors / cores), we are considering fully or partially heterogenous architecture.

Our goal is to construct an optimal schedule as to minimize the length of the periodic phase introduced in section 2.1. To do so, we also need to determine mapping for the actors to processors. The relation between mapping and scheduling was described in section 2.4.

Output for our algorithm will be a static schedule with the optimal criterion for the length of the period. In this schedule the actors will be statically mapped to particular processors. In this thesis, the variable $A_{i,j} = 1$ means actor a_i is mapped to processor $proc_j$. The schedule is divided into multiple time units. Because we are using multiple types of processors, we are proposing to use units which are universal in respect to the world clock e.g. nanoseconds.

We need to keep in mind that actors firing is nonpreemptive and channels with production rate P_k and consumption rate C_k constraints in our model are in terms of precedences. As mentioned in section 2.1, minimizing the length of the periodic phase is the same as maximizing the throughput because of inverse equation $Throughput = \frac{1}{Period}$ is true.

Because the mapping determines the length of the schedule we are facing a combinatorial explosion. For each mapping the schedule can be different and unique. We are looking for an optimal mapping. If we were to use brute force method on this problem, we would get a total of J^I possible mapping decisions. For example if we have 5 actors and 4 processors, we would get total of $4^5 = 1024$ possible mappings with possibility for 1024 different schedules with 1024 different period lengths.

Unfortunately, mapping decisions are not the only problem. Building the schedule is also difficult because of the constraints which are determined by consumption and production rates and the cyclic nature of the schedule and graph states.

Chapter 4

Baseline ILP

ILP formulation described in section 4.1 was introduced in [4]. The formulation was not complete and it needed to be fixed. Originally, the ILP did not contain any quantifiers. This ILP formulation does solve the given problem, but based on the experiments, it suffers from poor scalability. The results of this ILP can be found in Chapter 6. This ILP experiments will serve as a baseline for comparison with proposed improvements. To help the ILP with performance problems, we have extended this formulation for equations introduced in section 4.2 which slightly improves the ILP in terms of performance and more accurate schedule. This is because the ILP was originally targeting only the periodic phase, but as mentioned in sections 2.1 and 2.4, the transient phase is also important.

We are introducing table 4.1 which contains a description of the input variables and table 4.2 containing all decision variables used in the ILP formulation.

4.1 ILP Formulation

Formulation is trying to minimize the periodic phase in order to maximize the throughput. $Throughput = \frac{1}{Period}$ where $Period$ refers to overall length of the period in time units. In order to determine the length of the period, we need to build a schedule for particular mapping. Mapping and scheduling is strongly related to each other, therefore we need to determine the mapping and schedule together.

Implementation of the formulation was not given and we had to implement it into the IBM CPLEX framework. Equations described in this section are nonlinear. In order to use those constraints in the ILP framework, equations with multiplications of $A_{i,j}$ and $start(t)$ needed to be linearized. Linearization of the ILP is described in section 4.3.

In order to use this formulation we considered in section 2.3, we need to ensure the actor i is mapped only on one processor j. This is what equation (4.1) ensures.

$$\sum_{j \in J} A_{i,j} = 1 \qquad \forall i \in I \tag{4.1}$$

Table 4.1: Input variables

Variable name	Variable meaning
I	Set of actors
J	Set of processors
K	Set of channels between actors which set the precedence constraints containing: P_k - Number of produced tokens by the source actor on channel $k \in K$ C_k - Number of consumed tokens by the destination actor on channel $k \in K$ O_k - Number of the initials tokens on channel $k \in K$
$d_{i,j}$	Worst case execution time for actor $i \in I$ on Processor $j \in J$
T_{max}	Time estimation of the transient phase and the length of one period in the schedule.
n_i	Repetition vector for actor $i \in I$
$Period^{UB}$	Period upper bound
$Period^{LB}$	Period lower bound

Based on the SDF[3] model we are using introduced in section 2.1, we need to ensure that the target actor i_t can not be executed until it has all the data provided by the predecessor i_s. The indices in example SDF graph can be viewed in figure 4.1. In other words, equation (4.2) ensure that tokens produced by channel source actor a_{i_s} until time t plus initial tokens of channel k must be greater than all tokens consumed by the target actor a_{i_t} until time t.

$$C_k \cdot S_{i_t}(t) \leq P_k \cdot E_{i_s} + O_k \qquad \forall t \in \langle 0, T_{max} \rangle, \quad k \in K \tag{4.2}$$

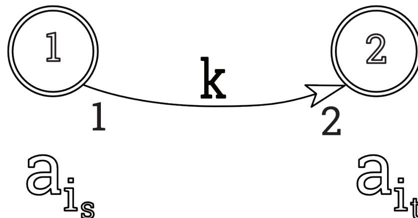


Figure 4.1: SDF graph with indexes introduced in equation (4.2)

While we need to count the number of started executions (S_i) of actor i and the number of ended executions (E_i), we are introducing equation (4.3) which anticipates that the firing of actors is not preemptive. Actor i has to start firing in time when

Table 4.2: Decision variables

Variable name	Variable meaning
$S_i(t)$	Number of started firings of actor $i \in I$ up to time $t \in 0...Tmax$
$E_i(t)$	Number of ended firings of actor $i \in I$ up to time $t \in 0...Tmax$
$A_{i,j}$	$A_{i,j} = \mathbf{1}$ Actor $i \in I$ is mapped to processor $j \in J$ $A_{i,j} = \mathbf{0}$ Otherwise
$start(t)$	$start(t) = \mathbf{1}$ Start of periodic phase is in time $t+1, t \in 0...Tmax$ $start(t) = \mathbf{0}$ Otherwise
$W_i(t)$	How much time is actor $i \in I$ executing up to time $t \in 0...Tmax$

it has ended firing decreased of its WCET i.e. $t_s = t_e - WCET_{i,j}$. Figure 4.2 shows how the variables S_i and E_i change based on time t in the schedule example.

$$S_i(t) = \sum_{j \in J} A_{i,j} \cdot E_i(t + d_{i,j}) \qquad \forall i \in I, \quad t \in \langle 0, T_{max} \rangle \tag{4.3}$$

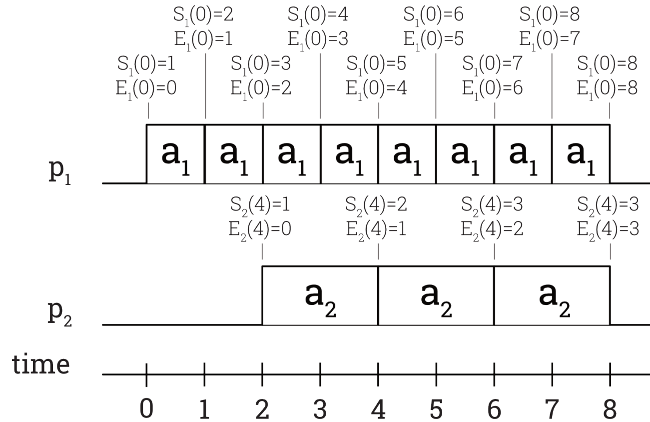


Figure 4.2: Schedule showing variables used by equation (4.3)

Equation (4.4) ensures that actor i is executed only once at a time. This equation is related to equation (4.1). Since actor i can be mapped only on one particular processor it can not be executing more then once in the same moment. This would mean one processor would be handling two tasks at the same, time which is not permitted in general.

$$\sum_{i \in I} A_{i,j} \cdot (S_i(t) - E_i(t)) \leq 1 \qquad \forall j \in J, \quad t \in \left\langle 0, T_{max} \right\rangle \tag{4.4}$$

Equation (4.5) makes sure that the amount of work done by actor i during the periodic phase is equal to the actors repetition vector n_i multiplied by its WCET $(n_i \cdot d_{i,j})$. Mapping decision $A_{i,j}$ makes sure we are counting WCET only for decided mapping. Counting the work done by actor i during the periodic phase is done by calculating the difference of total work done before the period starts $W_t(t) \cdot start(t)$ and total work done during the whole schedule $W_i(T_{max})$.

$$W_i(T_{max}) - \sum_{t=0}^{T_{max}} W_i(t) \cdot start(t) = n_i \cdot \sum_{j \in J} A_{i,j} \cdot d_{i,j} \qquad \forall i \in I \tag{4.5}$$

We are able to count work done (time while the actor i was executing) $W_i(t)$ by actor i in every time unit t by counting the difference between number of started $S_i(t)$ and finished $E_i(t)$ executions of actor i in time t. Difference between $S_i(t)$ and $E_i(t)$ can be in interval $\langle 0, 1 \rangle$, if and only if the difference is equal to 1 the task is actually executing in the time t. Equation (4.6) counts these differences and fills the vector $W_i(t)$.

$$W_i(t) = \sum_{t_2=0}^{t} (S_i(t_2) - E_i(t_2)) \qquad \forall i \in I, \quad t \in \left\langle 0, T_{max} \right\rangle \tag{4.6}$$

Since the periodic phase can be repeated infinitely, we are interested in the schedule of the transient phase and the schedule for one period, therefore we can introduce equation (4.7) which makes sure there is only one start of periodic phase $start(t)$ in the whole schedule.

$$\sum_{t=0}^{T_{max}} start(t) = 1 \tag{4.7}$$

Objective function (4.8) is minimizing the period. A period starts at time $t+1$, if and only if $start(t) = 1$. Minimizing the period has the same effect as maximizing the throughput due to inverse function $\left(Throughput = \frac{1}{Period}\right)$.

$$minimize \qquad \left\{ \quad Period = T_{max} - \sum_{t=0}^{T_{max}} start(t) \quad \right\} \tag{4.8}$$

4.2 Baseline ILP improvement

Part of this thesis is also an extension of the baseline ILP by equations introduced in this section. The objective function has been bounded by two equations, (4.9) and (4.11). Those equations dramatically decrease searching space of the solver which result in a greater solver performance. We are able to calculate the period lower bound $Period^{LB}$ and period upper bound $Period^{UB}$ before the solver begins because we use only input variables for the calculation i.e. equations (4.10) and (4.12) are not part of the ILP model and ther results can be considered as input variables.

$Period^{UB}$ is determined with the assumption that we have only one processor hence every actor needs to run sequentially and we assume the worst WCET for those actors (the worst mapping decision). $Period^{LB}$ is determined as length of the schedule if every actor runs in parallel and executes on the fastest resource available. Since every actor is mapped only to one processor and no actor can run simultaneously.

$$Period \leq Period^{UB} \tag{4.9}$$

$$Period^{UB} = \sum_{i \in I} \max_{j \in J} \{d_{i,j} \cdot n_i\} \tag{4.10}$$

$$Period \geq Period^{LB} \tag{4.11}$$

$$Period^{LB} = \max_{i \in I} \left\{ \min_{j \in J} \{d_{i,j} \cdot n_i\} \right\} \tag{4.12}$$

Because ILP formulation targets the period schedule it does not care about the transient phase. Therefore a schedule could start with $S_i > 0$ which would mean that some tasks would have started executing before time $t = 0$ which is obviously impossible. That is why we introduce equation (4.13).

$$S_i(0) + E_i(0) = 0 \qquad \forall i \in I \tag{4.13}$$

4.3 Linearization of the baseline ILP

In this section, we show how non-linear equations can be converted for usage in ILP frameworks. All non-linear equations (4.3 - 4.5) contain a multiplication of integer decision variables and binary decision variables. Hence, we can use linearization proposed in this section since the binary variable can contain only values $\in \{0, 1\}$. To use the linearization we need to define auxiliary variables X, Y, Z with usual indices where i, j stands for actor and processor respectively and t stands for time, we will use the variable M which is a sufficiently big enough integer.

Equation (4.3) can be linearized by equations (4.14 - 4.17). Those formulations can be expressed by condition: If actor i is mapped on processor j e.g. $A_{i,j} = 1$ then

auxiliary variable $X_{i,j}(t) = E_i(t+d_{i,j})$. This means that the sum introduced in (4.17) is the same as the sum as in the original equation (4.3).

$$X_{i,j}(t) \le A_{i,j} \cdot M \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.14}$$
$$X_{i,j}(t) \ge -M \cdot (1 - A_{i,j}) + E_i(t + d_{i,j}) \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.15}$$
$$X_{i,j}(t) \le M \cdot (1 - A_{i,j}) + E_i(t + d_{i,j}) \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.16}$$
$$S_i(t) = \sum_{j \in J} X_{i,j}(t) \qquad \forall i \in I, \quad t \in \langle 0, T_{max} \rangle \tag{4.17}$$

Another equation which needs to be linearized is equation (4.4). This is accomplished by constraints (4.18 - 4.21). The sum introduced in (4.21) counts the difference between S_i and E_i, if and only if the binary variable $A_{i,j}$ is equal to 1 i.e. if $A_{i,j} = 1$ then $Y_i(t) = S_i(t) - E_i(t)$. The original equation is again the same as (4.21).

$$Y_{i,j}(t) \le A_{i,j} \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.18}$$
$$Y_{i,j}(t) \le (S_i(t) - E_i(t)) \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.19}$$
$$Y_{i,j}(t) \ge A_{i,j} + (S_i(t) - E_i(t)) - 1 \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.20}$$
$$\sum_{i \in I} Y_{i,j}(t) \le 1 \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.21}$$

Last but not least, the equation which needed to be linearized is the constraint (4.5). Linearization of this equation is performed in the same way as in the examples before. We are using the auxiliary variable Z which is filled by condition: if $start(t) = 1$ then $Z_i(t) = W_i(t)$ which can be seen in equations (4.22 - 4.25) where (4.22) represents the original equation.

$$W_i(Tmax) - \sum_{t \in T}(Z_i(t)) = \\ = n_i \cdot \sum_{j \in J}(A_{i,j} \cdot d_{i,j}) \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.22}$$

$$Z_i(t) \le start(t) \cdot M \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.23}$$
$$Z_i(t) \ge -M \cdot (1 - start(t)) + W_i(t) \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.24}$$
$$Z_i(t) \le M \cdot (1 - start(t)) + W_i(t) \qquad \forall i \in I, \quad j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{4.25}$$

Chapter 5

Lazy constraints

Lazy constraints are constraints which can be added to ILP models during its solving process. It is useful when the model consists of a large number of constraints and it is not plausible for those constraints to be violated. The main motivation for using lazy constraints is to increase the performance in terms of solving time. But do not get mistaken because you can increase the performance by using only one lazy constraint. Also the solver can be slowed down if you use one badly chosen constraint. This is because the overhead for checking violations of these constraints and the process of adding the constraints to the model during the runtime can be much higher than having the constraints in the model from the very beginning.

Lazy constraints can be implemented in two ways by using IBM CPLEX Java application interface (API). Both choices offers their pros and cons which we discuss further in this chapter in sections 5.1 - 5.4.

5.1 Introduction to lazy constraints

In this section, we introduce lazy constraints usage in CPLEX Java API. We decided to use Java API because of its multi-platform usage. We will take a look into the lazy constraints function in subsection 5.1.1 and lazy callback class in subsection 5.1.2.

We consider both approaches because we would like to compare the difference between having full control over lazy constraints compared to leaving the responsibility up to the framework itself. Also the symmetry breaking algorithm introduced in section 5.3 cannot be achieved by the method proposed in subsection 5.1.1.

5.1.1 Lazy constraints function

Firstly, we would like to show how CPLEX can handle the lifecycle of those constraints. We only denote which constraints can be handled as lazy constraints and let CPLEX decide whether or not to add those constraints in the active model. Constraints denoted as lazy are placed in a pool of lazy constraints and are pulled out only if they are violated in a specific (last found) integer solution. The way CPLEX decides if the constraints are violated or not is simple. After CPLEX finds an integer

solution, it also checks if the lazy constraints placed in the pool have been violated or not by instating variables used in the constraint. This can be done because the CPLEX has the values of all of the variables (decision and input) by the time it finds an integer solution. After the lazy constraint is pulled from the pool, it is placed in the active model and the model continues to calculate. Of course the integer solution where the constraints have been violated is infeasible. The solver can also add the lazy constraints to the active model and place it back into the lazy constraints pool if the constraints have not been used for bounding the objective function for a while.

Switching from a common ILP model to a model which uses the lazy constraint callback functions is easier than using the callback described in section 5.1.2. Because the CPLEX cares about the lifecycle of the lazy constraints, we are freed from implementing whether or not the constraints need to be added into the model. CPLEX manages those decisions for us. Constraints can be added into ILP model using function ‘addLe‘, ‘addGe‘ and ‘addEq‘ for operations $\leq$, $\geq$ and $=$ respectively.

Code showing how equation (5.1) can be inserted into a model is shown in Code 5.1. This can easily be converted and the CPLEX can use a lazy constraint for the same equation as it is shown in Code 5.2.

$$left_expr \leq right_expr \tag{5.1}$$

```
1 cplex.addLe(
2     left_expr,
3     right_expr,
4     "equation_1"
5 );
```

Code 5.1: Add constraint to CPLEX model

```
1 cplex.addLazyConstraint(
2   (IloRange) cplex.le(
3     left_expr,
4     right_expr,
5     "equation_1"
6   )
7 );
```

Code 5.2: Add lazy constraint to CPLEX lazy constraints pool

Methods whose only function is adding lazy constraints has many pros. The most important part is that the CPLEX can handle the lifecycle of the constraints by itself. It usually is more effective than we would be doing it manually. Mainly because the CPLEX can take the lazy constraints from the active model and put it back into the lazy constraints pool. On the other hand, it cannot decide whether it should add the constraints during certain circumstances, e.g. based on the value of particular decision variables or a heuristic algorithm which uses the values of decision variables.

5.1.2 Lazy constraints callback

In order to use a more complex algorithm which will determine whether to add lazy constraints in the active model or not, we need to use lazy constraints callback. A callback is an abstract class in CPLEX Java API that can be extended. Its *main* function can be overridden by our custom implementation. This *main* function will run every time the integer solution is found.

Because we have access to all parameters, input and decision variables, we are able to decide whether or not to add the lazy constraints to the active model based on the values of the partial solution. Adding constraints to the active model can effect the solution and its feasibility. That means we need to be very careful when adding the constraints and our decision needs to be thoroughly justified.

Implementation of the lazy constraints callback is done by telling the CPLEX solver what class the CPLEX should consider when executing. The CPLEX decides what particular class should be used as the lazy constraints callback by its classtype. The best practice for lazy constraints callback is to have the implementing class of the lazy constraints callback as an inner class in Java. Because you have access to the decision variables from the callback, you do not have to pass these input variables nor decision variables to the callback class.

Code 5.3 shows how you can use the lazy constraints callback using the CPLEX Java API. The code in this example is the same as the code introduced in Code 5.2. But as you can see, we have more control in terms of adding the lazy constraints. The condition (line 16 in Code 5.3) which determines whether or not to add the lazy constraints in the active model (line 17 in Code 5.3) does not need to be in close relation to the constraint itself.

```java
public class ClassUsingLazyConstraintsCallback {

  void run() {
    cplex.use(new CustomLazyConstraintCallback());
    if (cplex.solve()) {
      printFeasibleSolution();
    } else {
      printInfeasibleSolution();
    }
  }

  private class CustomLazyConstraintCallback extends
      IloCplex.LazyConstraintCallback {

    @Override
    protected void main() throws IloException {
      if (equation_left_side >= equation_right_side) {
        this.add((IloRange) cplex.le(
          left_expr,
          right_expr,
          "equation_1"
        ));
      }
    }
  }
}
```

Code 5.3: Using lazy constraint callback in CPLEX Java API

5.2 ILP with lazy constraints

Experiments described in chapter 6 were initially performed over the baseline ILP model introduced in chapter 4. Based on the results of these experiments, we decided to improve the performance in terms of solving time because it was not possible to run realistic examples in a reasonable time. Since the model contains a large number of equations and variables, we decided to go with lazy constraints. This is also because we expected some of the constraints to be rarely violated or not essential to the model from the beginning while the solver starts.

By using lazy constraints, we expect the solving time will be lowered since fewer constraints will need to be verified and considered during the solving process and before the first integer solution is found. Even if the integer solution is found, it does not mean the lazy constraints were violated. We only add them if it is violated to keep the active model as small as possible.

We have chosen candidate equations (5.2 - 5.5) for the lazy constraints based on the assumption that equations which are not linear and contain a multiplication of decision variables. Decision variables used like this gluts solver, the solving time as

it increase with the number of equations. Also, we assume that equations which are related to the time (equations containing variable t) are overwhelming the solver. This is because $t \in \langle 0, T_{max} \rangle$ and T_{max} can be a large number. It grows with each applications repetition vector and actors WCET. This would exponentially increase the number of equations in the entire model.

Equation (5.2) was selected as the lazy constraint because it fills vector S on the basis of values contained in vector E i.e. it copies values from E to S only at proper time index $(t - d_{i,j})$. Equation (5.3) was chosen because we do not expect for it to be violated very often. The equation forbids multiple executions of actor i in time t. For example, where all the actors have self edge k with consumption rate equal or lower to production rate and initial tokens greater than those rates $C_k \leq P_k$ and $C_k \leq O_k$.

We need to mention that equations (5.2 - 5.5) were selected because they need to be linearized. This means we need to have 4 constraints in the model for each of these three constraints according the linearization proposed in section 4.3.

$$S_i(t) = \sum_{j \in J} A_{i,j} \cdot E_i(t + d_{i,j}) \qquad \forall i \in I, \quad t \in \langle 0, T_{max} \rangle \tag{5.2}$$

$$\sum_{i \in I} A_{i,j} \cdot (S_i(t) - E_i(t)) \leq 1 \qquad \forall j \in J, \quad t \in \langle 0, T_{max} \rangle \tag{5.3}$$

The results of using (5.2) and (5.3) were not significant so we have moved our focus onto the second candidates. Equations (5.4) and (5.5) were considered again because they contain variable t. This variable has a huge effect on prolonging the equation. We are assuming this equation wont be violated before the schedule is nearly complete because it strongly relates to the period which is at the end of the schedule.

$$W_i(T_{max}) - \sum_{t=0}^{T_{max}} W_i(t) \cdot start(t) = n_i \cdot \sum_{j \in J} A_{i,j} \cdot d_{i,j} \qquad \forall i \in I \tag{5.4}$$

$$W_i(t) = \sum_{t_2=0}^{t} (S_i(t_2) - E_i(t_2)) \qquad \forall i \in I, \quad t \in \langle 0, T_{max} \rangle \tag{5.5}$$

Experiments were performed multiple times with those equations. Firstly, we tried adding only one equation as the lazy constraint. There were performance improvements, but nothing that proved significant.

When we combined two lazy constraints equations, it resulted in a slightly improved performance. Based on the empirical results, our theory behind that is that when we add one constraints e.g. (5.4) the other constraint (5.5) that uses variable $W_i(t)$ is still present in the model. This shows that the ILP needs both bounded equations to be present in the model.

By combining three or more equations, with different variables, we recorded a lower performance. We decided to add one more equation which will be strongly related to the objective function. In expectation that the performance will increase. Equation (5.6) is related to the objective function because it limits only one periodic phase start in hopes this relation can improve the performance.

We thought that the performance would not increase by adding constraints that would be violated in most cases. Such as in equation (4.1) which ensures every actor will be mapped on one processor and equation (4.2) which ensures a correct firing of the actors regarding to the precedence constraints of the application model. This was experimentally proven.

$$\sum_{t=0}^{T_{max}} start(t) = 1 \tag{5.6}$$

The speedup of adding equation (5.6) as a lazy constraint with previously present lazy constraints (5.4) and (5.5) was significant. An important thing to point out is that while CPLEX uses lazy constraints callback, CPLEX computes only on one core by default. We can force CPLEX to run on multiple cores by switching one CPLEX parameter. This is introduced in section 5.4 but we did not have a good experience with it in terms of performance results. These and further results can be found in Chapter 6. We have experimented with both approaches. Lazy constraints callback was faster in smaller instances (instances with shorter solving times). On the other hand, lazy constraints function was better in terms of performance in bigger (slower solving times) instances. Our theory is that the solver overhead caused by maintaining the lazy constraints pool is more complex than the instance itself.

5.3 Symmetry breaking

Instances which are not meant for fully heterogenous systems can contain a lot of mapping symmetries. This means that entirely different mappings can result in the same schedule because WCET of actors are the same for different processors. It can be caused by processors which are the same type. Figure 5.1 shows how symmetrical schedules can look like. Because the actors a_1 and a_2 have the same WCET on processors p_1 and p_2 the schedule does not change while we change the mapping of $A_{1,1}, A_{2,2}$ to $A_{1,2}, A_{2,1}$. This means if we change the mapping of all of the actors from processor p_1 to processor p_2 and processors are the same type (all actors have same WCET on both processors) the length of the schedule must remain the same.

Symmetries can be seen in figure 5.2. Values which equal 0 i.e. $d_{i,j} = 0$ means the actor i cannot be mapped to processor j. In other words, actor i can be mapped only to processor j while $d_{i,j} \geq 1$. For example, two mappings which can be symmetrical are shown in figure 5.3. The difference is that we have flipped the mapping of actors with different processors of the same type with exactly the same WCET as before. In terms of scheduling, there is practically no difference.

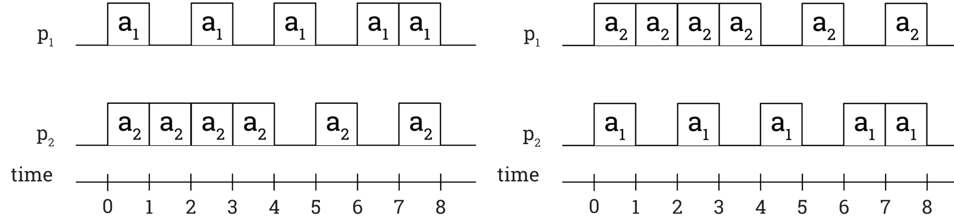


Figure 5.1: Symetrical schedules

$$d = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 2 \end{pmatrix}$$

Figure 5.2: Matrix of WCET with symmetries

The schedule for A^1 will most likely be the same as the schedule for A^2. The solver can determine the execution of actors sequences with little difference and not violate the constraints while the length of the period remains the same. This is because the model does not distinguish the processors in any other way than by execution times for particular actors. This is also because communication between processors is not considered in this particular model. Otherwise, a mapping could result in a longer communication overhead while the other would not.

Since the symmetries are based on the WCET of actors assigned to specific processors, all of the symmetries can be determined from the input variables. To do this, we propose an algorithm which finds all the symmetries in section 5.3.1. Later, in subsection 5.3.2 we discuss how to use this information to remove symmetries from the solution.

5.3.1 Algorithm to find symmetries

Algorithm 1 takes input $d_{i,j}$ which consists of the WCET of all of the actors assigned to a single processor. In every step, it compares two rows to find the symmetries

$$A^1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad A^2 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 5.3: Symmetrical mapping

by freezing one row (row^1) of d matrix and compares the values with the other row (row^2). It is comparing two columns of these rows (col^1 and col^2). If all those values in the selected rows and columns match, the information that processors have the same WCET for those actors is recorded to symmetry matrix Sym at index Sym_{row^1,row^2} and also diagonally symmetrical Sym_{row^2,row^1}. While we read the symmetrical matrix and $Sym_{i_1,i_2} \neq \emptyset$ to mean there are symmetries between actors i_1 and i_2 on the processors listed in values recorded to the matrix Sym at index i_1, i_2.

```
[1]  Input: d_{i,j}
[2]  Output: Set of Symmetries S
[3]  i1 = 0;
[4]  for i1 < d.length do
[5]  |   row^1 = d[i1];
[6]  |   i2 = i1 + 1;
[7]  |   for i2 < d.length do
[8]  |   |   row^2 = d[i2];
[9]  |   |   j1 = 0;
[10] |   |   for j1 < row^1.length do
[11] |   |   |   col^1 = {row^1[j1], row^2[j1]}
[12] |   |   |   if col^1[0] == 0 OR col^1[1] == 0 then
[13] |   |   |   |   continue;
     |   |   |   end
[14] |   |   |   j2 = j1 + 1;
[15] |   |   |   for j2 < row^1.length do
[16] |   |   |   |   col^2 = {row^1[j2], row^2[j2]}
[17] |   |   |   |   if col^1[0] == 0 OR col^1[1] == 0 then
[18] |   |   |   |   |   continue;
     |   |   |   |   end
[19] |   |   |   |   if col^1[0] == col^2[0] == col^1[1] == col^2[1] then
[20] |   |   |   |   |   Sym[i1][i2].add({j1, j2});
[21] |   |   |   |   |   Sym[i2][i1].add({j1, j2});
     |   |   |   |   end
[22] |   |   |   |   j2++;
     |   |   |   end
[23] |   |   |   j1++;
     |   |   end
[24] |   |   i2++;
     |   end
[25] |   i1++;
     end
     return Symmetries
```

Algorithm 1: Finding symmetries

We propose an algorithm which is able to find all symmetries based on the input containing WCET of each actor on each processor. As mentioned before the input

can contain values equal to 0 i.e. $d_{i,j} = 0$.

Algorithm 1 shows the basic implementation in pseudocode on how it is possible to find symmetries. Becouse the size and complexity of the problem is based on building the schedule, the algorithm which finds the symmetries in matrix d can be simple in terms of complexity. Because we do not expect instances with a huge number of nodes or processors, this algorithm will run only once before CPLEX begins its solving phase.

Our algorithm can find symmetries only when the WCET of actors involving symmetry mapping is exactly the same. This is because we cannot be sure that the symmetrical mapping with different WCET will not affect the schedule.

Symmetry matrix for WCET matrix d introduced in figure 5.2 can result after using our algorithm 1 into a matrix which look likes the matrix (Sym) in figure 5.4. As you can see, regarding algorithm 1, the Sym matrix containing indices of actors Sym_{i_1,i_2} vector - pair, which means what processors are symmetrical to the actor respectively. For example, $Sym_{2,3} = \{2,3\}$ is holding information that actors 3 and 4 are symmetrical on processors 2 and 3 index starting at 0.

$$d_{i,j} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 2 \end{pmatrix} \qquad Sym = \begin{pmatrix} \emptyset & \{0,1\} & \emptyset & \emptyset \\ \{0,1\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{2,3\} \\ \emptyset & \emptyset & \{2,3\} & \emptyset \end{pmatrix}$$

Figure 5.4: Symmetries for matrix introduced on figure 5.2

5.3.2 Symmetry breaking using lazy constraints callback

Since we are adding lazy constraints to active models based on their values from the first integer solution, we are not able to add those constraints into the lazy constraints pool before the solver starts. Therefore we need to use lazy constraints callback as we discussed in section 5.1.2. The main idea for this approach is to remove all branches of the ILP tree which contains symmetrical solutions from the one which had been found.

Lets say the integer solution will find a mapping which can be seen in figure 5.5. We are able to express this mapping by a number of equations (5.7). Using this mapping, we are able to build an equation which forbids symmetrical mapping.

Constraints which break symmetries is mathematically described in equations (5.8 - 5.11). These equations forbid symmetrical mapping. While the solution consists of mapping decisions introduced in figure 5.5, we are forbidding any other symmetrical mapping. Symmetrical mapping would, for example $\boldsymbol{A_{1,2} = 1;}$ $\boldsymbol{A_{2,1} = 1;}$ $A_{3,3} = 1;$ $A_{4,4} = 1$, violate the lazy constraints (5.8) and (5.9). Also, please note that it is forbidden to map two actors on the same processor e.g. $A_{1,1} = 1;$ $\boldsymbol{A_{2,1} = 1;}$ $A_{3,3} = 1;$ $A_{4,4} = 1$ because the objective function can not be any better in terms of length of

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad A_{1,1} = 1; \qquad A_{2,2} = 1; \qquad A_{3,3} = 1; \qquad A_{4,4} = 1 \tag{5.7}$$

Figure 5.5: Initial mapping and symmetries for the instance

the period. The best case would be the actor with a changed mapping would fill the empty spaces in the schedule. This would result in the same length of the schedule and periodic phase would remain the same length. This is an example for the input matrix $d_{i,j}$ containing WCET of actors and symmetry matrix Sym, both shown in figure 5.4.

$$A_{1,1} + A_{1,2} + A_{2,1} + A_{3,3} + A_{4,4} \leq 3 \tag{5.8}$$
$$A_{1,2} + A_{2,1} + A_{2,2} + A_{3,3} + A_{4,4} \leq 3 \tag{5.9}$$
$$A_{3,3} + A_{3,4} + A_{4,3} + A_{1,1} + A_{2,2} \leq 3 \tag{5.10}$$
$$A_{3,4} + A_{4,3} + A_{4,4} + A_{1,1} + A_{2,2} \leq 3 \tag{5.11}$$

We introduce three functions in Code 5.4. The function *removeSymmetries* accept decision variable matrix A, which represents mapping, and matrix of symmetries Sym as input. The algorithm does not provide any output because the main purpose of the algorithm is for it to add the lazy constraints to the active model based on the input variables.

If the algorithm finds that the mapping is determined in index i, j ($A_{i,j} = 1$) at line 4, it starts searching if there are any symmetries for actor i present in the model. If there is a symmetry (line 6) between actors i and js, the algorithm checks whether or not actor js is mapped to the symmetrical processor $j2$ (line 10). While the actors i and js are not mapped to the same processor, it adds the equations which forbids this symmetrical mapping by creating the lazy constraints found on line 15. If those actors are mapped to the same processor, it forbids the mapping of both actors to the other processor by adding the constraints referred to on line 17.

The function $addTerm(c, var)$ will add term $(c \cdot var)$ to the expression while function $addEqutionLe(Expression[]\ left, Expression\ right)$ will add the equation $left \leq right$ to the model.

The algorithm select one mapped actor as "active". The equation for symmetry breaking was created as a copy of the current mapping without the "active" actor and its symmetries. For example, we selected $A_{1,1}$ as "active". Its symmetrical actor mapping is $A_{2,2}$ so the equation will be $A_{3,3}+A_{4,4}$. Then, we need to add a symmetrical mapping to our $A_{1,1}$ and $A_{2,2}$ with one of those actors. Hence, the final equation will be $A_{1,2} + A_{2,1} + A_{2,2} + A_{3,3} + A_{4,4}$ or $A_{1,2} + A_{2,1} + A_{1,1} + A_{3,3} + A_{4,4}$. This created equation must be lower or equal to the number of mapped actors decreased by 1.

```
void removeSymmetries(A, Sym) {
  int[][] X = new int[A.length][A[0].length];
  for (int i = 0; i < A.length; i++) {
    for (int j = 0; j < A[i].length; j++) {
      if (A[i][j] == 1) {
        for (int js = i+1; js < Sym[i].length; js++) {
          if (Sym[i][js] != null) {
            sMap = -1;
            for (int j2 = 0; j2 < A[js].length; j2++) {
              if (A[js][j2] == 1) {
                sMap = j2;
              }
            }
            if (sMap != j) {
              breakCrossSymmetries(A, i, j, js, sMap);
            } else {
              breakColumnSymmetries(A, i, j, js, sMap);
            }
          }
        }
      }
    }
  }
}

void breakCrossSymmetries(A, i, j, js, sMap) {
  Expression currentMappingEx = new Expression();
  int c = 0;
  int[][] T = A;
  T[i][ Sym[i][js][0] ] = 0;
  T[i][ Sym[i][js][1] ] = 0;
  T[js][ Sym[i][js][0] ] = 0;
  T[js][ Sym[i][js][0] ] = 0;

  for (int a = 0; T.length; a++) {
    for (int b = 0; T[a].length; b++) {
      if (T[a][b] == 1) {
        currentMappingEx.addTerm(1, A[a+1][b+1]);
      }
    }
  }

  Expression breakSymm = new Expression();
  breakSymm.addTerm(1, A[i+1][ Sym[i][js][0]+1 ]);
  breakSymm.addTerm(1, A[i+1][ Sym[i][js][1]+1 ]);
  breakSymm.addTerm(1, A[js+1][ Sym[i][js][0]+1 ]);
  breakSymm.addTerm(1, A[js+1][ Sym[i][js][1]+1 ]);

```

```
49    breakSymm.addTerm(-1, A[i+1][ j+1 ]);
50
51    model.addEqutionLe({
52      currentMappingEx,
53      breakSymm,
54    },
55    A.length - 1
56    );
57
58    Expression breakSymm2 = new Expression();
59    breakSymm2.addTerm(1, A[i+1][ Sym[i][js][0]+1 ]);
60    breakSymm2.addTerm(1, A[i+1][ Sym[i][js][1]+1 ]);
61    breakSymm2.addTerm(1, A[js+1][ Sym[i][js][0]+1 ]);
62    breakSymm2.addTerm(1, A[js+1][ Sym[i][js][1]+1 ]);
63
64    breakSymm.addTerm(-1, A[js+1][ sMap+1 ]);
65
66    model.addEqutionLe({
67      currentMappingEx,
68      breakSymm,
69    },
70    A.length - 1
71    );
72
73
74
75  }
76
77  void breakColumnSymmetries(A, i, j, js) {
78    Expression currentMappingEx = new Expression();
79    int c = 0;
80    int[][] T = A;
81    T[i+1][ Sym[i][js][0]+1 ] = 0;
82    T[i+1][ Sym[i][js][1]+1 ] = 0;
83    T[js+1][ Sym[i][js][0]+1 ] = 0;
84    T[js+1][ Sym[i][js][0]+1 ] = 0;
85
86    for (int a = 0; T.length; a++) {
87      for (int b = 0; T[a].length; b++) {
88        if (T[a][b] == 1) {
89          currentMappingEx.addTerm(1, A[a+1][b+1]);
90        }
91      }
92    }
93
94    Expression breakSymm = new Expression();
95    if (symmetryMapping != Sym[i][js][0]) {
96      breakSymm.addTerm(1, A[i+1][ Sym[i][js][1]+1 ]);
```

```
97      breakSymm.addTerm(1, A[js+1][ Sym[i][js][1]+1 ]);
98    } else {
99      breakSymm.addTerm(1, A[i+1][ Sym[i][js][0]+1 ]);
100     breakSymm.addTerm(1, A[js+1][ Sym[i][js][0]+1 ]);
101   }
102   model.addEqutionLe({
103     currentMappingEx,
104     breakSymm,
105   },
106   A.length - 1
107   );
108 }
```

Code 5.4: Functions used in callback of CPLEX Java API to break symmetries

5.4 CPLEX parameters affecting performance

Disabling cuts Since we are focusing on the performance upgrade of the proposed baseline ILP, we should also target the CPLEX itself. We have found that several parameters can affect the CPLEX solver performance and solving time. W believe that parameters can affect different ILP formulations in different ways. To determine this, we are proposing to use experimental testing for any particular formulation. Cuts commented in Code 5.5 were not helpful and did not improve speed up time.

All the parameters are disabling cuts in CPLEX. More about CPLEX parameters can be found in ILOG CPLEX 11.0 Parameters Reference Manual [12] or in ILOG CPLEX 11.0 User's Manual [11]. The value of those parameters are set at 0 by the default. This means the CPLEX will decide if cuts should be generated or not. There is no explanation in the documentation about how these decisions are made. Documentation says, we quote: "Setting the value to 0 (zero), the default, indicates that the attempt to generate cuts should continue only if it seems to be helping."

```
1 cplex.setParam(IloCplex.IntParam.DisjCuts, -1);
2 cplex.setParam(IloCplex.IntParam.FracCuts, -1);
3 cplex.setParam(IloCplex.IntParam.LiftProjCuts, -1);
4 // cplex.setParam(IloCplex.IntParam.MCFCuts, -1);
5 cplex.setParam(IloCplex.IntParam.MIRCuts, -1);
6 cplex.setParam(IloCplex.IntParam.ZeroHalfCuts, -1);
```

Code 5.5: CPLEX cut parameters

Lazy constraints callback multithreading Since the default implementation of CPLEX uses only one thread for computation, the usage of lazy constraints callback can be slowed down because it does not use the full potential of the machine. This can be affected by setting the number of threads which CPLEX should use, as you can see in Code 5.6. This parameter is also useful when you need to use only a part

of the capacity of the machine. We have also experimented using lazy constraints callback with all of the CPU threads available.

```
cplex.setParam(IloCplex.IntParam.Threads, 4);
```

Code 5.6: CPLEX cut parameters

Chapter 6

Experimental Setup

In this chapter, we introduce instances which have been used for experimental testing. Experiments were performed in several setups. Baseline ILP was benchmarked in IBM OPL Studio as well as in Java implementation with the goal to compare these two implementations. Then, we introduce benchmark results for lazy constraints and lazy constraints callback as well as the symmetry breaking algorithm used in lazy constraints callback.

We define what instances were used for our experiments in section 6.1, where you will find several example graphs. All graphs can be found in Appendix A. We also introduce a table comparing real application sizes. You can find the specification of the hardware used in the experiments in section 6.2. In sections 6.3 and 6.4 we compare the differences between an implementation of the baseline ILP introduced in section 4.1. We have implemented the baseline ILP in IBM OPL Studio and Java with use of CPLEX Java API. Experiments in sections 6.5 and 6.6 introduce the results of the two different approaches using the lazy constraints and the symmetry breaking algorithm.

6.1 Benchmark instances

Benchmark instances used for comparison of implementation in Java were selected from the SDF^3 tool. Our goal was to experiment with commonly known instances and also with real application models. That way we can compare our approach with others in terms of performance.

We have found that the size of the SDF graph expressed by the number of nodes (actors) and edges (channels) do not have a significant effect on the problem's (instance) complexity in terms of performance. In many cases, it seems like the rates are more important because it determines the length of the schedule (value of T_{max}) which has a larger impact on solving time. Rates corresponding to the size of the HSDF are listed in table 6.1. With more actors present in the HSDF, the periodic phase of the schedule consists of more actors firing. This results in a longer period and longer constraints present in the model.

For example, instance of the H263 decoder which look small at first (shown in figure 6.1) is more complicated to solve using the ILP formulation than instance of the modem in figure 6.2.

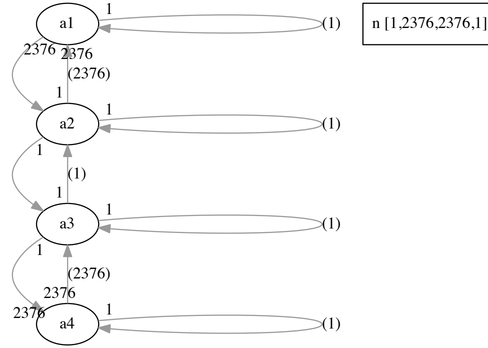


Figure 6.1: H263 Decoder with buffer

Table 6.1: Size of the real application instances used for benchmarking

	Number of actors		Number of channels	
Instance name	SDF	HSDF	SDF	HSDF
Satellite receiver	22	4515	74	18723
Sample-rate converter	6	612	16	2654
MP3 decoder	13	13	37	37
Modem	16	48	54	170
H.263 decoder	4	4754	10	19010
MP3 decoder (fully heterogenous)	4	4754	10	19010

Graphs used in this thesis for benchmarking have been attached in Appendix A. We have faced performance problems with the SDF^3 benchmark instances due to the particular problems complexity. At the beginning, we had to set a reasonable solving time which the experiment should finish. This deadline for completion was set after the first experiments at 2 days (48 hours). This breakpoint was satisfied in most instances, but instance of the H263 decoder were not solved by the proposed ILP in the given time.

We are introducing several results in this chapter. If any tables containing results have the X sign instead of a proper time, it means the solver did not finished within the 48 hour limit. We also propose a CPU time in some cases. The CPU time is the sum of times of every thread used by the solver and the solving time refers to the real time which humans can perceive. This metric would become very relevant in the case the implementation of CPLEX would handle parallel processing a better way in using lazy constraints. Current implementation stop the other threads while the callback is executing and needs to wait until the lazy constraints callback finishes.

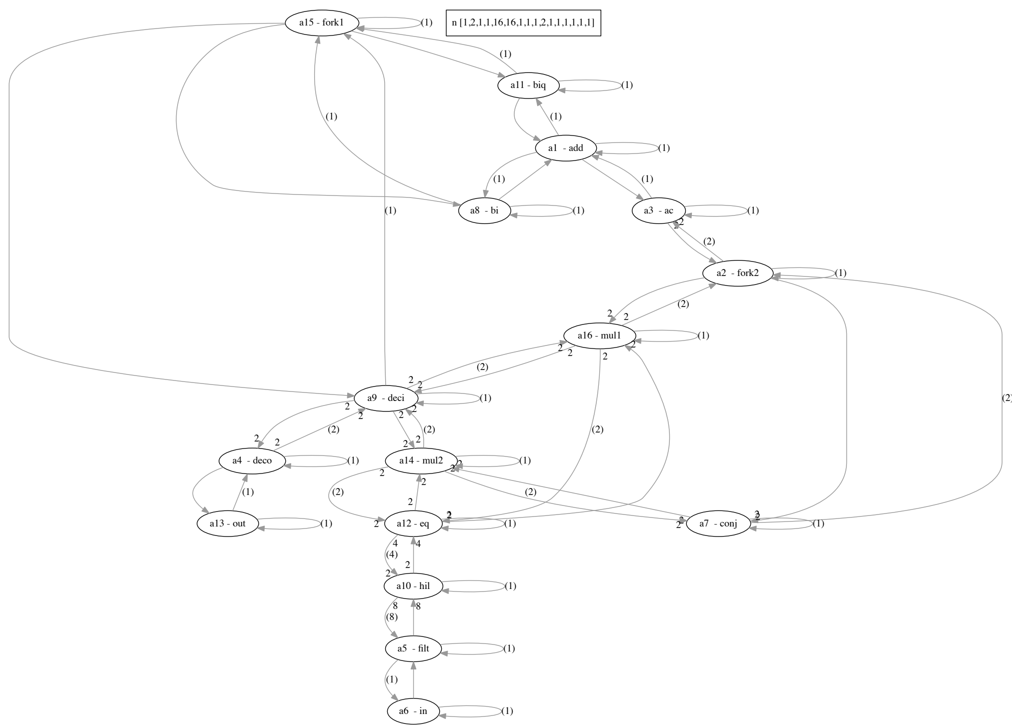


Figure 6.2: Modem with buffer

6.2 Hardware

Several experiments used server Kepler based in the CTU in Prague, Faculty of Electical Engineering and a Mac Book Pro laptop. Specifications for both devices can be found in table 6.2

Table 6.2: Specification of hardware used for experiments

Kepler server	
CPU:	Intel Xeon E5-2620 v2 @ 2.1GHz / Turbo Boost up to 2.6GHz / 6 cores
RAM:	64 GB
Laptop (Mac Book Pro)	
CPU:	Intel Core i5 @ 2.4GHz / Turbo Boost up to 3,1 GHz / 4 cores
RAM:	8 GB

Experiments introduced in sections 6.3, 6.4 and 6.5 were performed on the "Kepler" server. After series of experiments, the Kepler server had been allocated to different research and we tested the symmetries on a Mac Book Pro laptop. Instances that used the Kepler server were unsolvable for the laptop in a reasonable time and we had to use another set of instances. This was not an issue because the instances used on Kepler server did not contain any symmetries. For the symmetry breaking algorithm, a new set of instances were generated by the SDF^3 and adapted to contain a different number of symmetries.

6.3 Baseline experiments using IBM OPL Studio

OPL Studio was used for its relatively fast implementation. It served as our baseline benchmark which we used later on to compare our experiments. Table 6.3 shows the results of experiments executed on the Kepler server while table 6.1 offers an overview of the complexity of the problem. It seems like the size of the HSDF does matter in terms of problem complexity. Size of the HSDF is related to the production and consumption rates of the channels.

The considered platform for instances listed in table 6.1 was partially heterogenous, which means actors were able to fire on multiple processors (not all of the processors were considered due to their performance issues). The input matrix d, for MP3 decoder instance can be seen in figure 6.3 with a different WCET. All experiments using instances from table 6.1 were executed on the Kepler server specified in section 6.2.

6.4 Baseline ILP Java experiments

To understand what the effect is of rewriting the same algorithm from the IBM OPL Studio to Java using CPLEX Java API, we prepared the exactly same set of benchmarks for a Java implementation. The results can be seen in table 6.3.

$$d = \begin{pmatrix} 0 & 19 & 0 & 0 & 0 & 32 & 0 & 0 & 37 & 0 & 21 & 0 & 0 \\ 20 & 0 & 19 & 0 & 0 & 37 & 0 & 0 & 32 & 0 & 0 & 21 & 0 \\ 0 & 83 & 0 & 0 & 0 & 0 & 85 & 0 & 48 & 0 & 0 & 0 & 0 \\ 132 & 0 & 0 & 0 & 132 & 0 & 0 & 145 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 89 & 0 & 0 & 0 & 0 & 80 & 83 & 0 & 0 \\ 132 & 0 & 0 & 132 & 0 & 0 & 0 & 0 & 0 & 80 & 83 & 80 & 87 \\ 0 & 0 & 68 & 0 & 68 & 0 & 0 & 0 & 69 & 0 & 0 & 0 & 0 \\ 0 & 0 & 68 & 0 & 0 & 0 & 0 & 69 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 2 \\ 0 & 0 & 34 & 0 & 42 & 0 & 0 & 0 & 0 & 0 & 0 & 47 & 0 \\ 0 & 9 & 0 & 0 & 7 & 0 & 0 & 12 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 36 & 0 & 0 & 0 & 0 & 42 & 0 & 0 & 0 & 0 & 5 \\ 4 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$

Figure 6.3: Input of partially heterogenous instance instance of MP3 decoder

Table 6.3: Results of experiments using Kepler server for calculation

	IBM OPL Studio	Java CPLEX API	Lazy constraints
Instance name	Solving time [s]	Solving time [s]	Solving time [s]
Satellite receiver	2247.14	X	X
Sample-rate converter	212.66	X	X
MP3 decoder	53.62	54.78	25.26
Modem	0.38	0.58	0.54
H.263 decoder	X	X	X
MP3 decoder (Fully heterogenous)	X	2408	322

Also, to further prove our point that the lazy constraints can speed up the solving process, we have created a new, fully heterogenous instance of a MP3 decoder with an input showen in figure 6.4. This instance is used in experiments using Java API and in experiments using the lazy constraints for comparison. Also, it is essential to mention that the instance is man made. This is because the benchmark instance did not provide us with realistic input for particular architecture.

We are not certain what caused a huge difference in some instances and no change in others. We assume the solver core implementation in Java and IBM OPL Studio can vary and the solver would choose a different branching approach with result of different solving time.

6.5 Baseline ILP with lazy constraints

Our goal was to make the ILP faster, which was achieved by adding the lazy constraints introduced in sections 5.1.1 and 5.1.2. We have introduced two results, table

$$d = \begin{pmatrix} 19 & 19 & 32 & 21 & 37 & 32 & 21 & 37 & 37 & 32 & 21 & 32 & 41 \\ 20 & 18 & 19 & 36 & 37 & 37 & 18 & 20 & 32 & 37 & 36 & 21 & 43 \\ 84 & 83 & 83 & 85 & 28 & 66 & 85 & 92 & 48 & 57 & 94 & 45 & 136 \\ 132 & 148 & 132 & 196 & 132 & 145 & 139 & 145 & 165 & 178 & 212 & 113 & 245 \\ 87 & 67 & 56 & 98 & 89 & 134 & 156 & 192 & 174 & 80 & 83 & 123 & 89 \\ 132 & 92 & 78 & 132 & 98 & 76 & 89 & 217 & 134 & 80 & 83 & 80 & 87 \\ 96 & 97 & 68 & 45 & 68 & 39 & 74 & 96 & 69 & 38 & 94 & 88 & 114 \\ 78 & 77 & 68 & 68 & 69 & 37 & 78 & 69 & 49 & 57 & 98 & 45 & 78 \\ 6 & 10 & 2 & 5 & 12 & 1 & 1 & 4 & 1 & 5 & 1 & 9 & 2 \\ 9 & 12 & 34 & 7 & 42 & 45 & 87 & 67 & 56 & 12 & 19 & 47 & 21 \\ 12 & 9 & 36 & 9 & 7 & 76 & 35 & 12 & 46 & 5 & 89 & 6 & 9 \\ 51 & 46 & 36 & 92 & 122 & 89 & 90 & 42 & 48 & 9 & 85 & 27 & 5 \\ 4 & 1 & 9 & 1 & 18 & 19 & 1 & 1 & 21 & 1 & 2 & 67 & 89 \end{pmatrix}$$

Figure 6.4: Input of fully heterogenous instance instance of MP3 decoder

6.3 contains the same instances from sections 6.3 and 6.4 which have been calculated on the Kepler server.

It is important to point out that the smaller difference between the $Period^{LB}$ and the actual objective function was the better performance upgrade lazy constraints provided. This is caused by the fact that if we bind the objective function, which minimizes the period and solution with the lowest possible period is found, the solver can stop its execution.

The results of the first three experiments described in sections 6.3, 6.4 and 6.5 are graphically displayed in figure 6.5 The chart is made from the real time (world) clock, and is clearly shown that the lazy constraints increase the performance of the CPLEX solver with a factor at least 2. This resulted in a significant speedup.

Our conclusion for these experiments is that rewriting this particular formulation into CPLEX Java API did not increased the performance. Generally the CPLEX Java API seems to slow things down. But by using lazy constraints in Java, which had significant performance upgrades in all instances, which was able to be solved in Java without usage of lazy constraints. We assume the Java API can approach the branching and cutting strategies differently, therefore the performance can vary in different instances.

6.6 Symmetry breaking experiments

In this section, we introduce a new set of experiments which are tested in several ways. The main purpose was to have baseline results which we can use and compare to our results for the symmetry breaking algorithm. All the instances were generated by the SDF^3 tool.

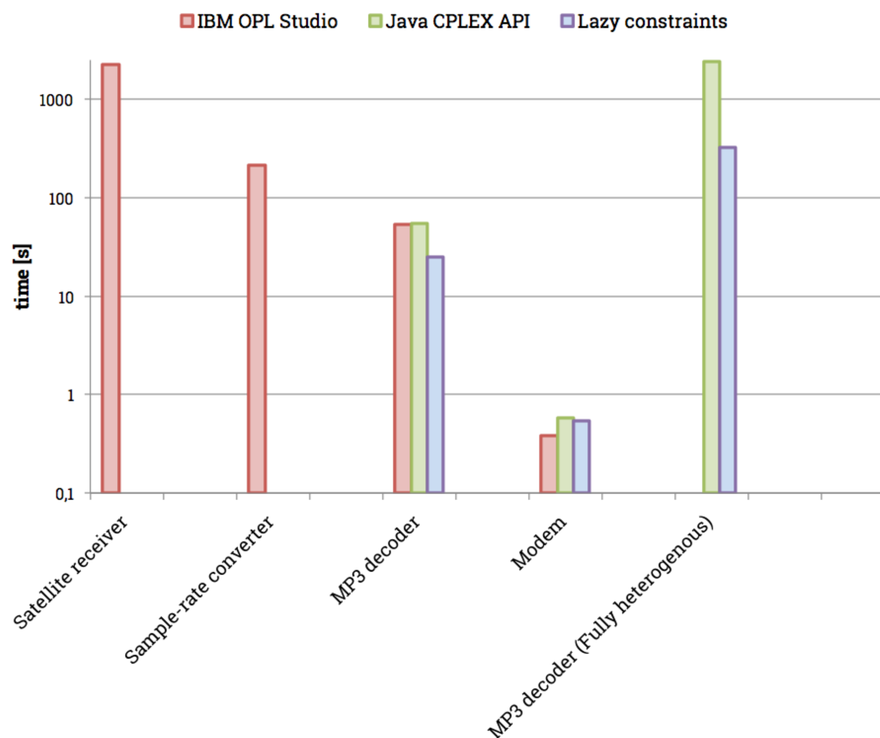


Figure 6.5: Solving time of first instance set in real time

Table 6.4 contains the results from our various experiments. In the prepared set of instances, we found that the lazy constraints method approach can increase the solver performance, but only in terms of CPU time. Also, the experiments show that instances ”Modem”, ”Artificial instance 4” and ”Artificial instance 10”, which does not contain any symmetries, are slowed down by the symmetry breaking algorithm because the algorithm does not reduce the searching space. The algorithm executes every time the solver finds an integer solution. On the other hand, instances ”Artificial instance 2” and ”Artificial instance 8” recorded CPU time performance improvements.

The disadvantage of using the lazy constraints is that the CPLEX solver can only use one thread while running. Or to be more specific, more threads can be forced by parameter, but we did not have any good results doing so. It is probably caused by the callback itself, because every other branching needed to be stopped before the callback had finished executing.

Figures 6.6 and 6.7 displays the values of solving times in real (world) clock time and CPU time respectively. From those figures, it is clear that in most cases lazy constraints save CPU time, but in world clock time, are still outperformed due to a lack of multithreading computation.

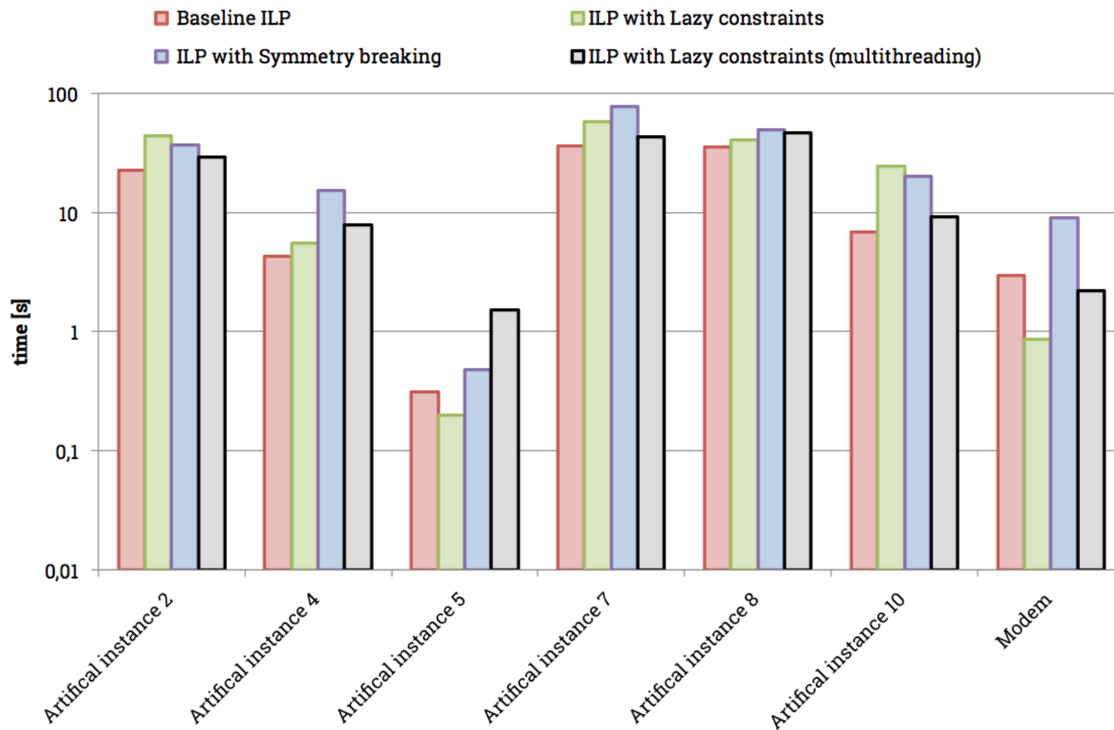


Figure 6.6: Solving time of second instance set in real time

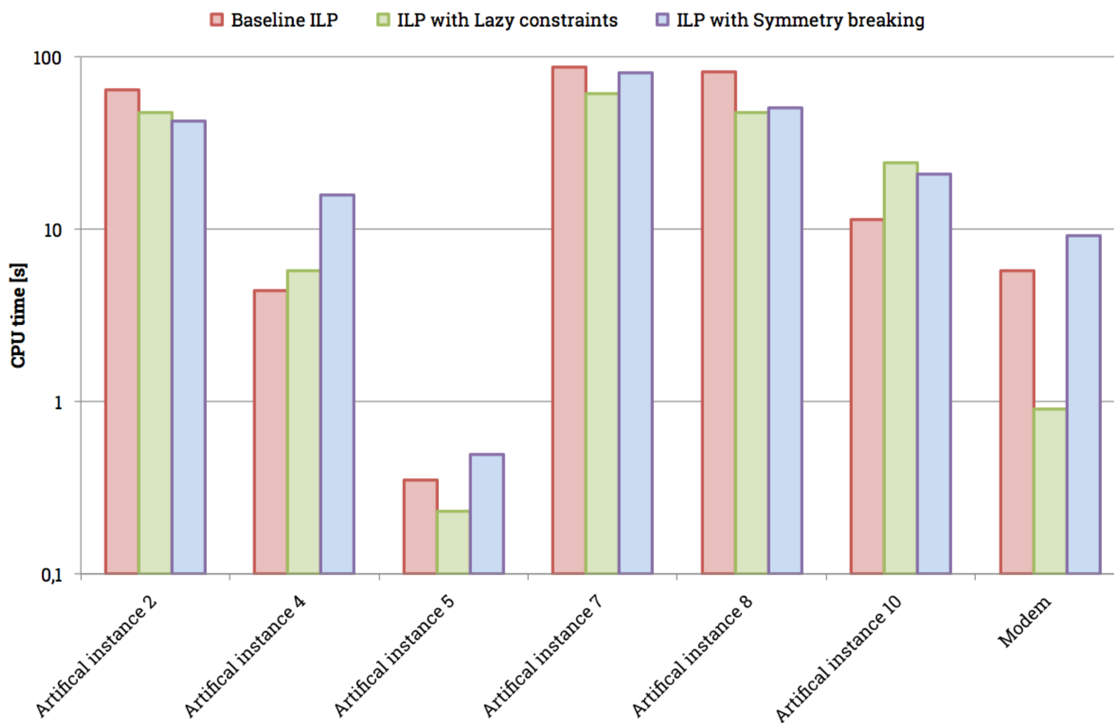


Figure 6.7: Solving time of second instance set in CPU time

Table 6.4: Results of the experiments using lazy constraints on MacBook Pro

Instance name	Baseline ILP time [s]	Baseline ILP CPU time [s]	ILP with Lazy constraints time [s]	ILP with Lazy constraints (multithreading) time [s]	ILP with Lazy constraints CPU time [s]	ILP with Symmetry breaking time [s]	ILP with Symmetry breaking CPU time [s]
Artifical instance 2	22.802	**64.42**	44.074	29.457	**47.57**	37.166	**42.5**
Artifical instance 4	4.322	**4.399**	5.576	7.903	**5.763**	15.416	**15.75**
Artifical instance 5	0.312	**0.350**	0.200	1.514	**0.231**	0.479	**0.491**
Artifical instance 7	36.074	**87.690**	58.114	42.935	**61.270**	78.048	**81.048**
Artifical instance 8	35.349	**81.630**	41.129	46.38	**47.429**	49.412	**50.493**
Artifical instance 10	6.866	**11.350**	24.293	9.249	**24.458**	20.035	**20.997**
Modem	2.970	**5.75**	0.859	2.192	**0.908**	9.079	**9.246**

Chapter 7

Related works

In this chapter, we present several works which have been done in field of application mapping and scheduling. We mainly focus on real-time applications. Each work listed approaches our problem, a similar problem, or approaches it from a different angle. We are providing a quick overview of these approaches.

[10] describes how we can represent real-time data driven applications using SDF graphs. Also, how computation of the throughput is done. Unfortunately it does not cover the problem of mapping and scheduling. [7] does not deal with the mapping or scheduling, but it shows what needs to be considered if we modeled a real system with communication between processors and shared memories. [9] is a great survey that includes both what has been done in the past and what the modern trends are with a brief introduction to the selected problems. The survey also considers a lot of works for non-real-time systems which are not completely relevant to our problem.

Different models for streaming applications are proposed in [5]. They assume the application is a directed acyclic graph (DAG) . This is similar to HSDF which are not allowed to have cycles. HSDF is a special case of SDF graph where every actor fires only once per graph iteration / period. This is a restriction compared to what we consider. For the computation, they considered the mapping and scheduling part with communication over FlexRay bus between multiple applications. For computation, they decided to use an ILP. The disadvantages of their approach compared to this thesis is that they firstly decide the mapping and then maked the schedule for the feasible mapping. This does not guarantee optimality at first place. Moreover, they only assume homogenous platforms.

A constraint programming (CP) approach is explained in [8]. They also propose an algorithm for conversion from SDF graph to "Rate Homogeneous SDF" where every actor fires once, but not every rate is equal to one. In practical use, this is the same as using a HSDF. They also consider the communication by adding two actors that send and receive messages between processing units. On the other hand, they do not describe the whole CP model, but just the necessary usage of variables. In addition, they only use small instances of problems, but are able to map and schedule multiple applications running simultaneously to satisfy the real-time constraints on each of them.

A way to find energy efficient mapping and schedule is proposed in [14]. However, they use heuristics for both mapping and scheduling. They also only consider homogeneous systems. This is the same system architecture that we consider, but we are more general with the architecture. They consider two different power states for chips and memories. They try to find a minimal energy consuming schedule while satisfying the throughput requirement. It was done by slowing down the processors or by changing the size of the memories. These calculations are done with usage of a ILP model. The downside to their approach was that they were not able to ensure optimality of the schedule or the mapping because they used heuristics.

[6] deals with scheduling and mapping in a different manner. They use time division multiple access (TDMA) to schedule the actors and build a static schedule from it. They use CP for the scheduling. Every time a partial schedule is found, the maximum cycle mean (MCM) algorithm runs to check if this solution is still feasible. Their objective function is to maximize processors utilization subject to minimum average throughput constraints. This means they are targeting at different goal while using a different optimization criterion.

A description of how the mapping and scheduling can be done using only an ILP formulation is presented in [4]. This formulation is introduced in section 4.1. Since the formulation seems reasonable, we decided to use it as our baseline ILP. Unfortunately, they still omit a few things in their formulation. They do not propose any quantifiers and their formulation is non-linear. It is easy to linearize these equations and we show how it can be done in 4.3 for usage in an ILP framework. An interesting part about this paper is that they try to minimize the linear combination of several real-time constraints, such as the length of the periodic phase, latency and cost. They propose using pareto fronts as to find all suitable solutions. Each solution has another impact on the throughput, processor cost function and latency. That paper has an extension [3] published later on where the ILP formulation is extended with communication aware mapping. This means that we have a reference for our experiments, which we can compare with. The baseline ILP served as starting point for our research.

Articles which explain the basic usage of lazy constraints for area harvesting problem is introduced in [2]. Lazy constraints are meant to reduce the computation time of the solver, which is related to our work. We used this article as a starting point for the lazy constraints. It briefly describes how the lazy constraints can be used and how they work. Unfortunately, it does not describe how to implement lazy constraints and what the possible options are. This was described in this thesis.

Chapter 8

Conclusion and Future work

Our main goal was to find an algorithm that can map actors to processors and build a static schedule in a reasonable time. A successful mapping and scheduling algorithm was introduced in the form of an ILP formulation which had been implemented in IBM OPL Studio as well as in Java using CPLEX Java API as a point of comparison.

The baseline ILP had several issues which needed to be solved in order for it to be implemented into a CPLEX framework. Linearization of non-linear equations was briefly described in section 4.3.

The complexity of this problem is due to the mapping and scheduling which has to be done simultaneously because every mapping decision can have an effect on the schedule. We have described the needs for this in section 2.4 Also, the complexity is given by size of the instances. While the instances do not seem complex at first, the SDF graph is complex because presence of the precedence constraints in the form of consumption and production rates.

The drawbacks of the ILP formulation are clear. It has serious performance issues. To solve it, we have implemented lazy constraints into the model and proposed a symmetry breaking algorithm. While the lazy constraints have good results in most cases, the symmetry breaking algorithm increases the performance only when a large number of symmetries are present in the input. We assume this is because the overhead of the lazy constraints callback and lack of multithreading. Because the CPLEX needs to pause solving and wait until lazy constraints callback is finished every time the callback is invoked.

We had proven that fully heterogenous instance of MP3 decoder can be solved more effectively with the usage of lazy constraints. We have accomplished a speedup with a factor of more than 7. On the other hand, the symmetry breaking algorithm did not have a significant speedup. We assume this was caused by the overhead of the lazy constraints callback together with the inability to use multiple threads effectively.

We have partially solved the problem in some cases, but it still remains open for a large number of instances since we have not found solution for performance issues in general. We have proven the problem is not trivial for real applications and it is not largely scalable for now. Also, the provided solution does not cover real architecture because it is missing essential components e.g. interprocessor communication and finite memories. Hence, we would suggest to continuing with further performance

upgrades. We think, it would be worth trying a heuristic brute force algorithm for mapping. Once we have the mapping decided, the ILP formulation can be simplified for equations which needs to be linearized.

It is important to find a way to determine the mapping and static schedule for applications with communication such as in network on chip (NoC). Real systems have to communicate between processors, which need to be scheduled. This was done in [3], but since it is an extension of the ILP proposed in this thesis, it would most probably suffer from the same performance issues as the baseline ILP. As we have proven, not all the real application models provided by the SDF^3 would execute in a reasonable time using the baseline ILP formulation e.g. H.264 decoder with a buffer is too large for modern computers and the calculation take up a lot of memory and time.

Also, the mapping of data to memories used in the architecture is an important extension of this ILP. Real systems have different types of memories with different sizes and access types. Exploring this can enable applications to execute more efficiently. Decisions about this are not trivial and does not necessary means "if it fits in local memory use it as data storage". The problem is that there is a lot of data sitting in a local memory wasting space before it is actually needed.

Things that are worth mentioning is that finding mapping and schedule in such a way that we minimize the energy consumption on the model with multiple power modes of processors and find out how those units need to be configured to ensure the real-time constraints and minimize energy consumption. This would have a greater use in any industry by prolonging the battery life and also during an energetic crisis, which the world is currently in. Power efficiency is becoming an important design phase a is why heterogenous architectures are becoming so popular.

Bibliography

[1] M. Geilen. Reduction techniques for synchronous dataflow graphs. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 911–916, July 2009.

[2] N. Könnyű and S. F. Tóth. A cutting plane method for solving harvest scheduling models with area restrictions. *European Journal of Operational Research*, 228(1):236–248, 2013.

[3] J. Lin, A. Gerstlauer, and B. L. Evans. Communication-aware heterogeneous multiprocessor mapping for real-time streaming systems. *Journal of Signal Processing Systems*, 69(3):279–291, 2012.

[4] J. Lin, A. Srivatsa, A. Gerstlauer, and B. L. Evans. Heterogeneous multiprocessor mapping for real-time streaming systems. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1605–1608. IEEE, 2011.

[5] M. Lukasiewycz and S. Chakraborty. Concurrent architecture and schedule optimization of time-triggered automotive systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 383–392. ACM, 2012.

[6] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 57–66. ACM, 2007.

[7] A. Nelson, K. Goossens, and B. Akesson. Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor soc. *Journal of Systems Architecture*, 2015.

[8] K. Rosvall and I. Sander. A constraint-based design space exploration framework for real-time applications on mpsocs. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 326. European Design and Automation Association, 2014.

[9] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013.

[10] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, pages 777–782. ACM, 2007.

[11] Ilog cplex 11.0 user's manual | cuts description.
http://www-eio.upc.es/lceio/manuals/cplex-11/html/usrcplex/solveMIP14.html, state to 20. 4. 2015.

[12] Ilog cplex 11.0 parameters reference manual.
http://www-eio.upc.es/lceio/manuals/cplex-11/pdf/refparameterscplex.pdf, state to 20. 4. 2015.

[13] Sdf3 tool | homepage, 2015.
http://www.es.ele.tue.nl/sdf3/, [Online; accessed 20. April 2015].

[14] J. Zhu, I. Sander, and A. Jantsch. Energy efficient streaming applications with guaranteed throughput on mpsocs. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 119–128. ACM, 2008.

Appendix A

Graphs used for benchmarking

In this appendix we are providing a set of inputs. A subset of them were used in our thesis for experiments. Nodes in the graph represents different actors. Their names are written in each of the nodes. Production and consumption rates are present on the edge beside each actor. In the middle of the edge can be present initial token in the parenthesis. Also, we must mention that if no number of production or consumption rate is present, the rate is equal to 1. The repetition vector is always listed in the rectangle beside each graph.

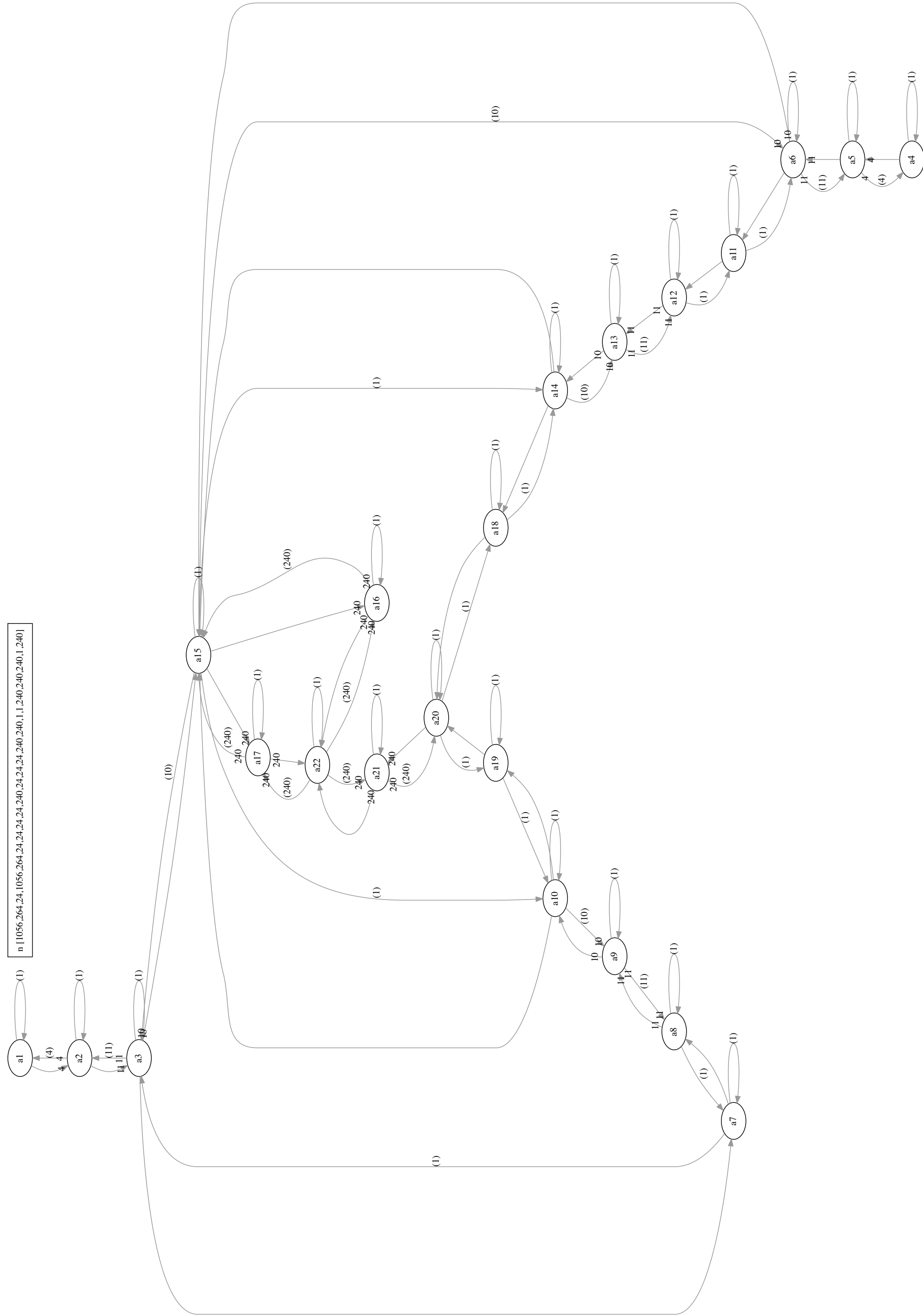


Figure A.1: Satellite receiver with buffer

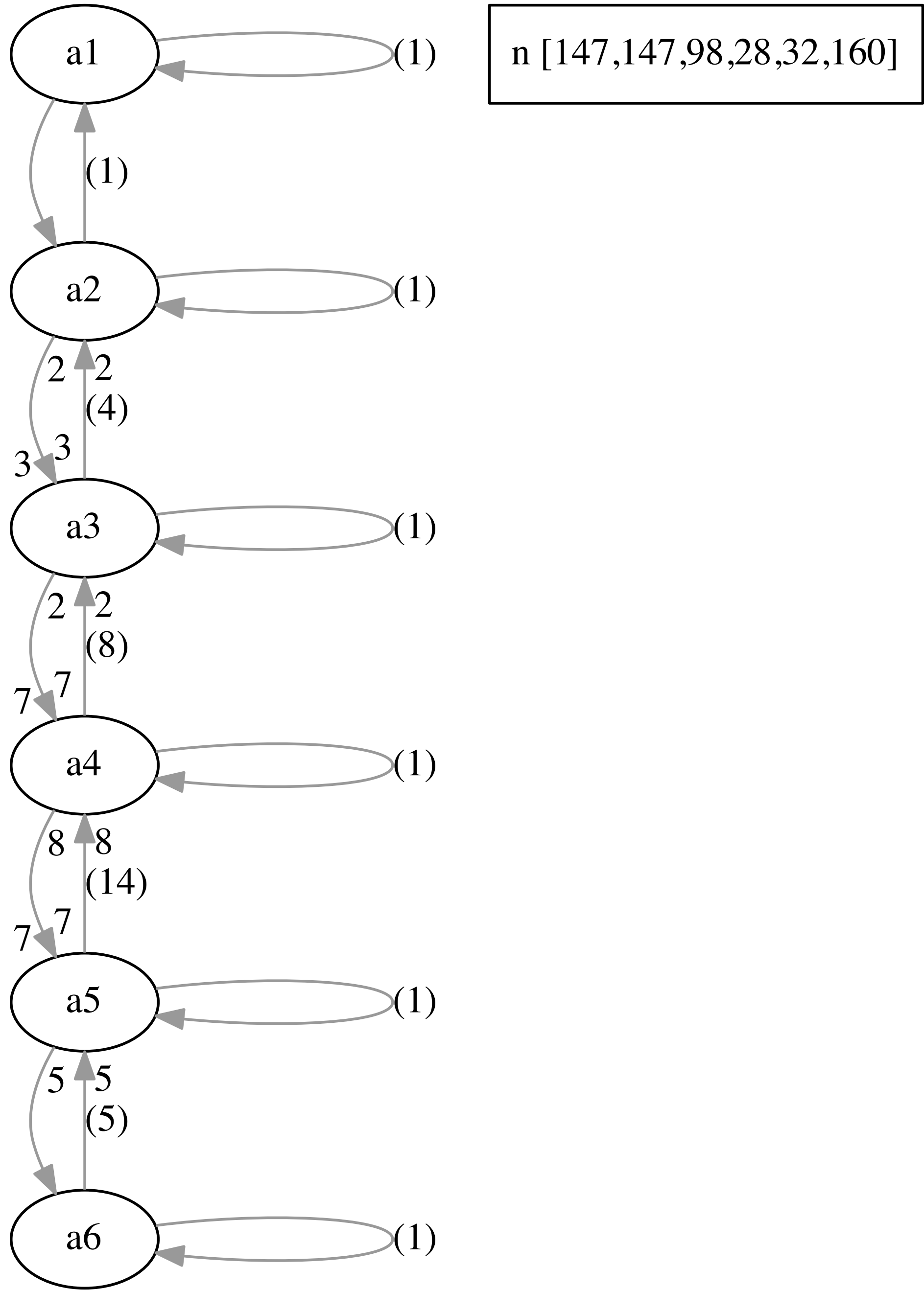


Figure A.2: Sample-rate converter with buffer

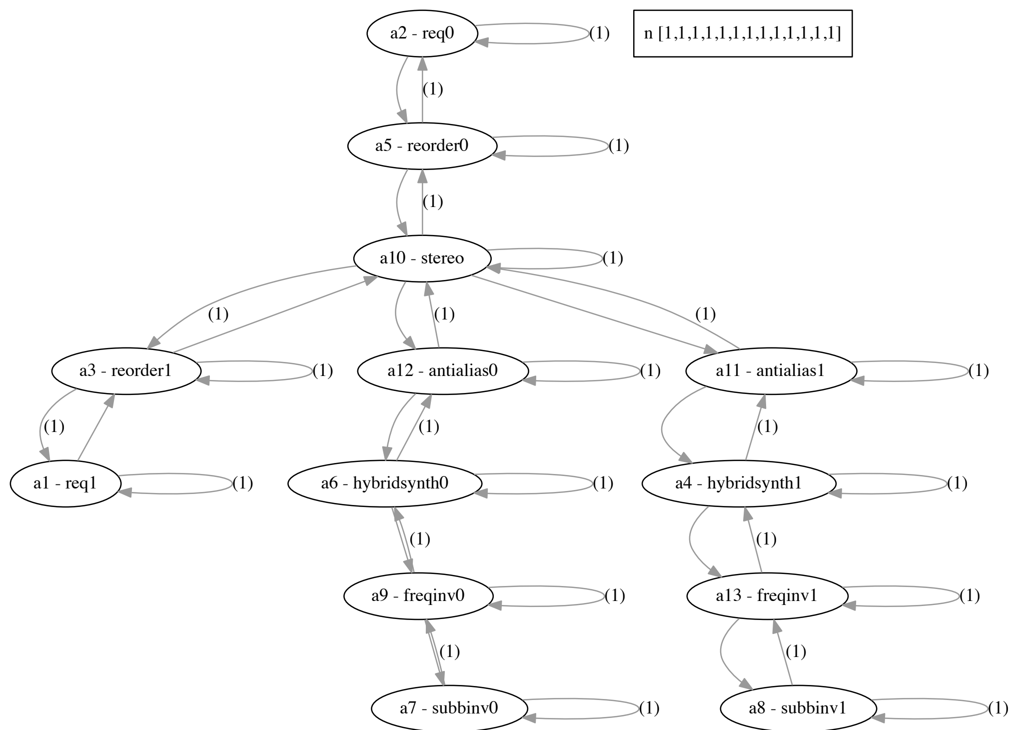


Figure A.3: MP3 decoder with buffer

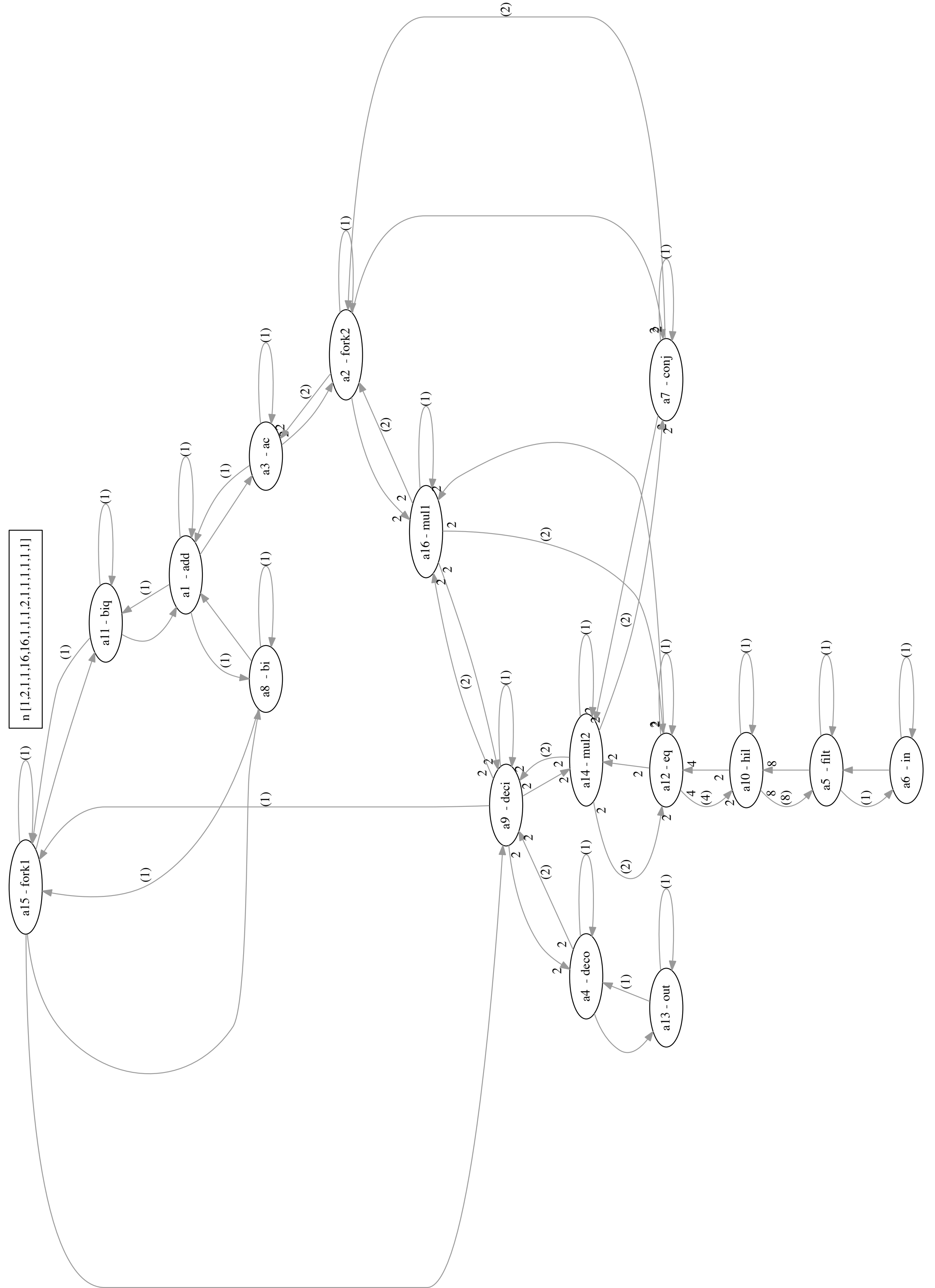


Figure A.4: Modem with buffer

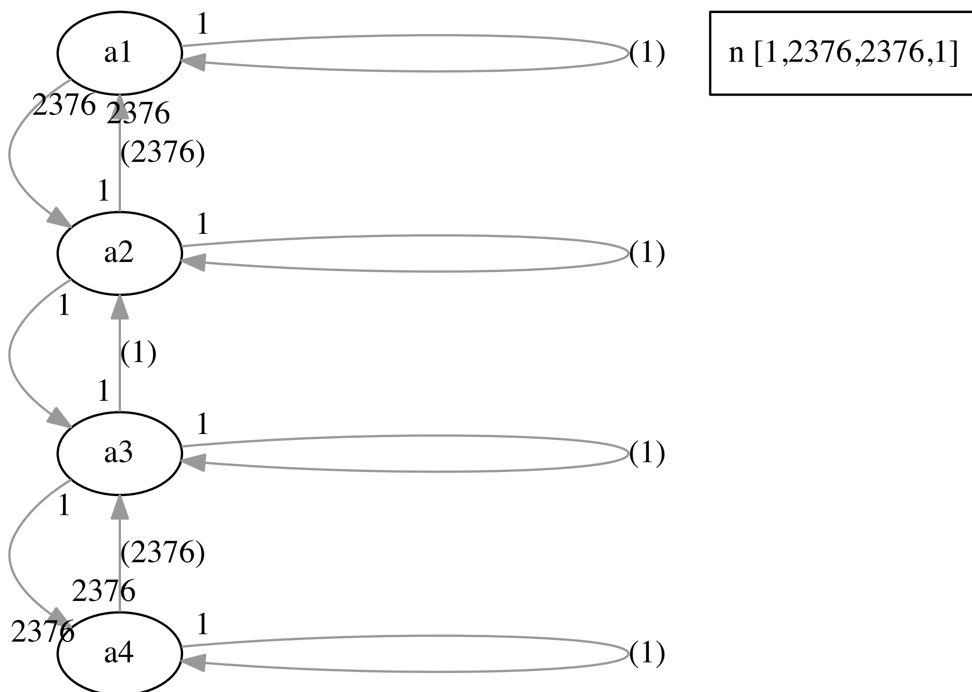


Figure A.5: H.263 decoder with buffer

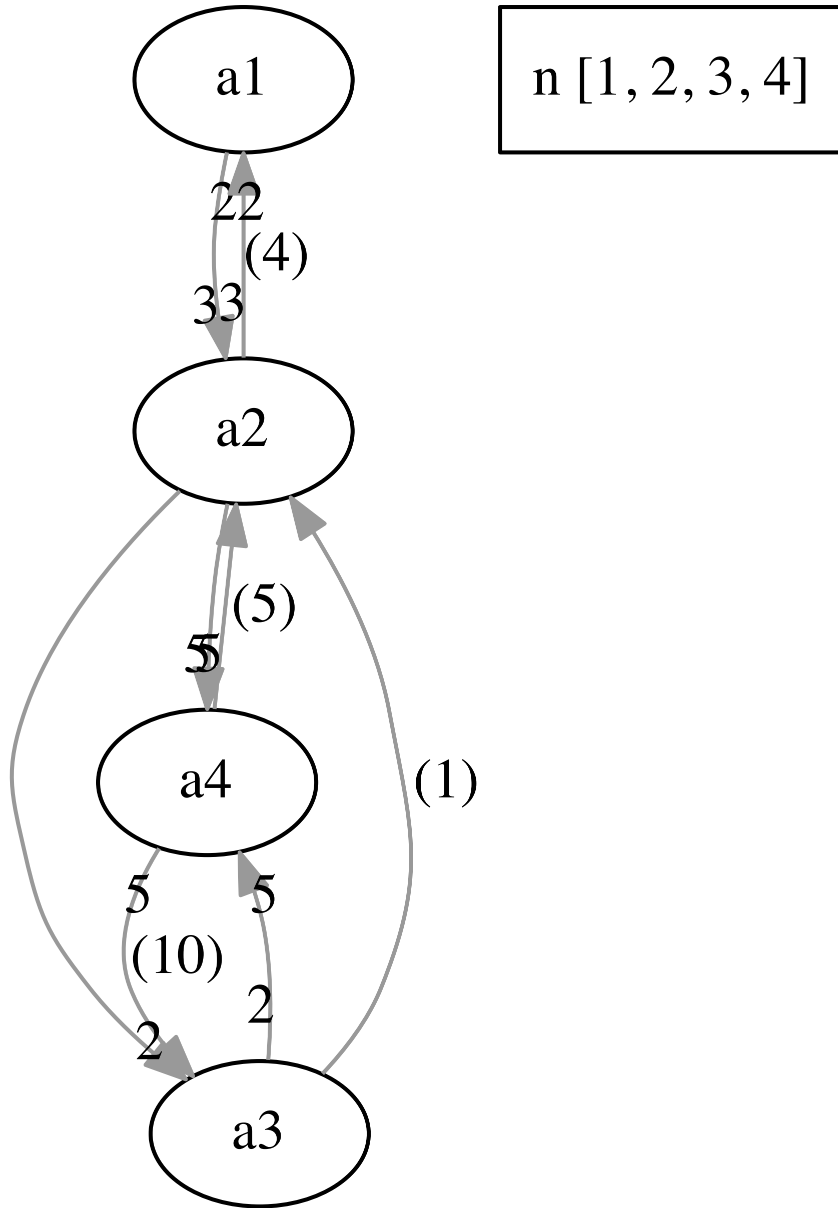


Figure A.6: Artifical instance 1

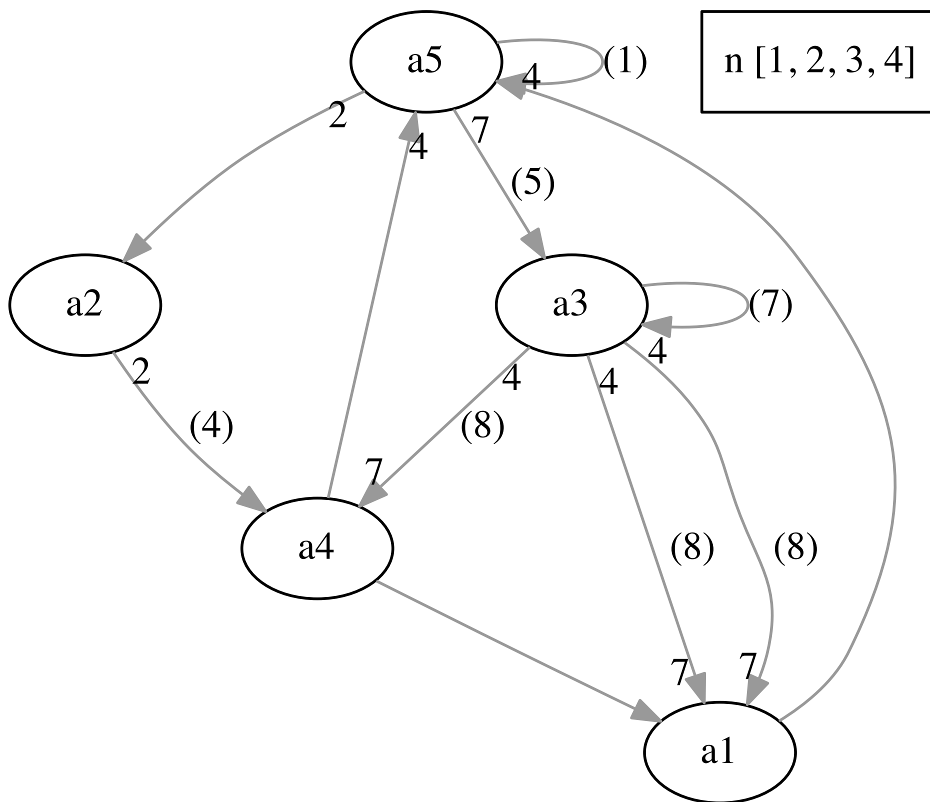


Figure A.7: Artifical instance 2

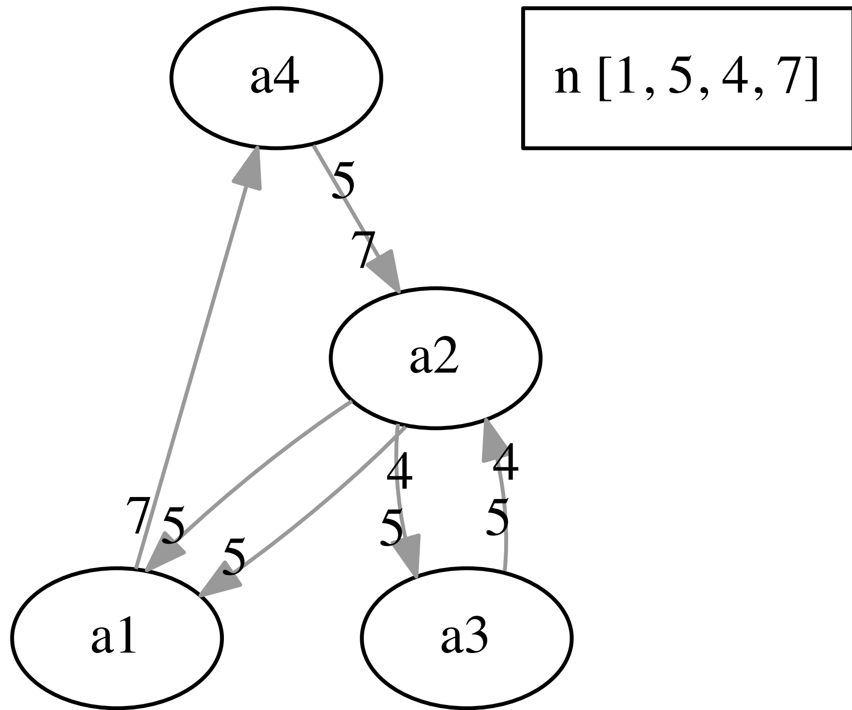


Figure A.8: Artifical instance 3

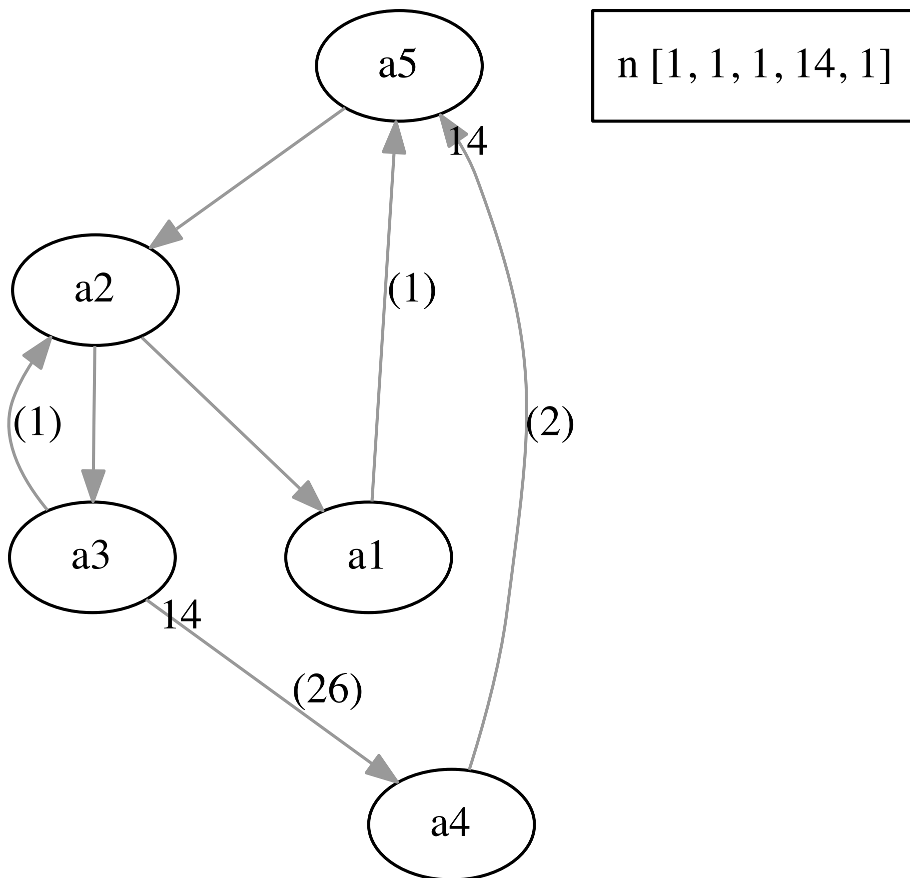


Figure A.9: Artifical instance 4

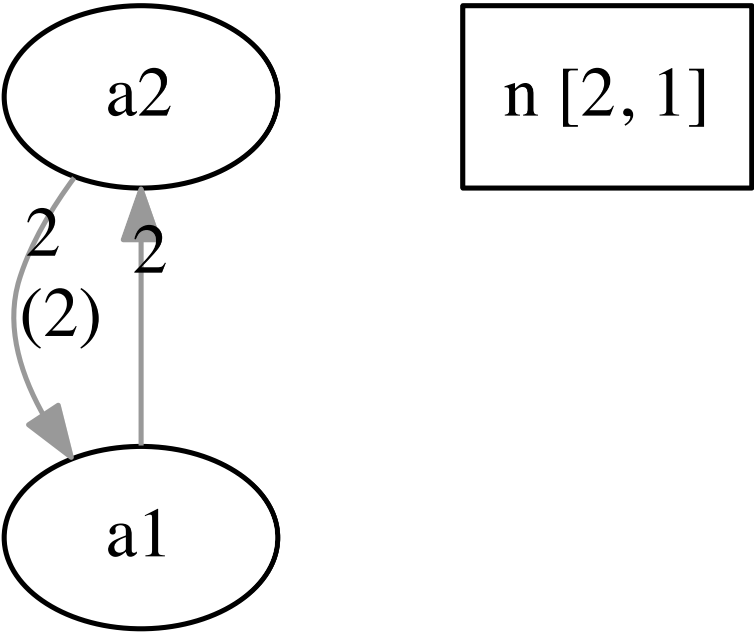


Figure A.10: Artifical instance 5

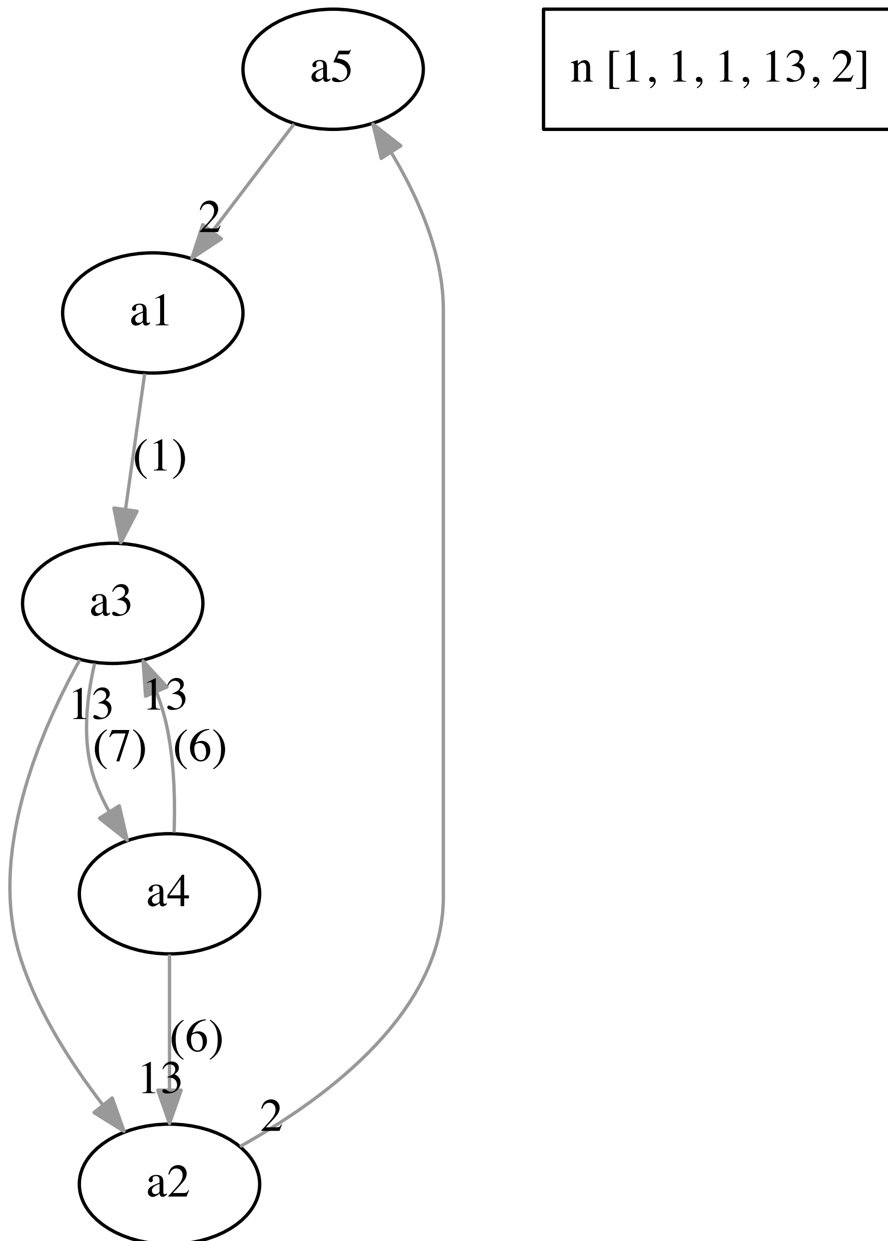


Figure A.11: Artifical instance 6

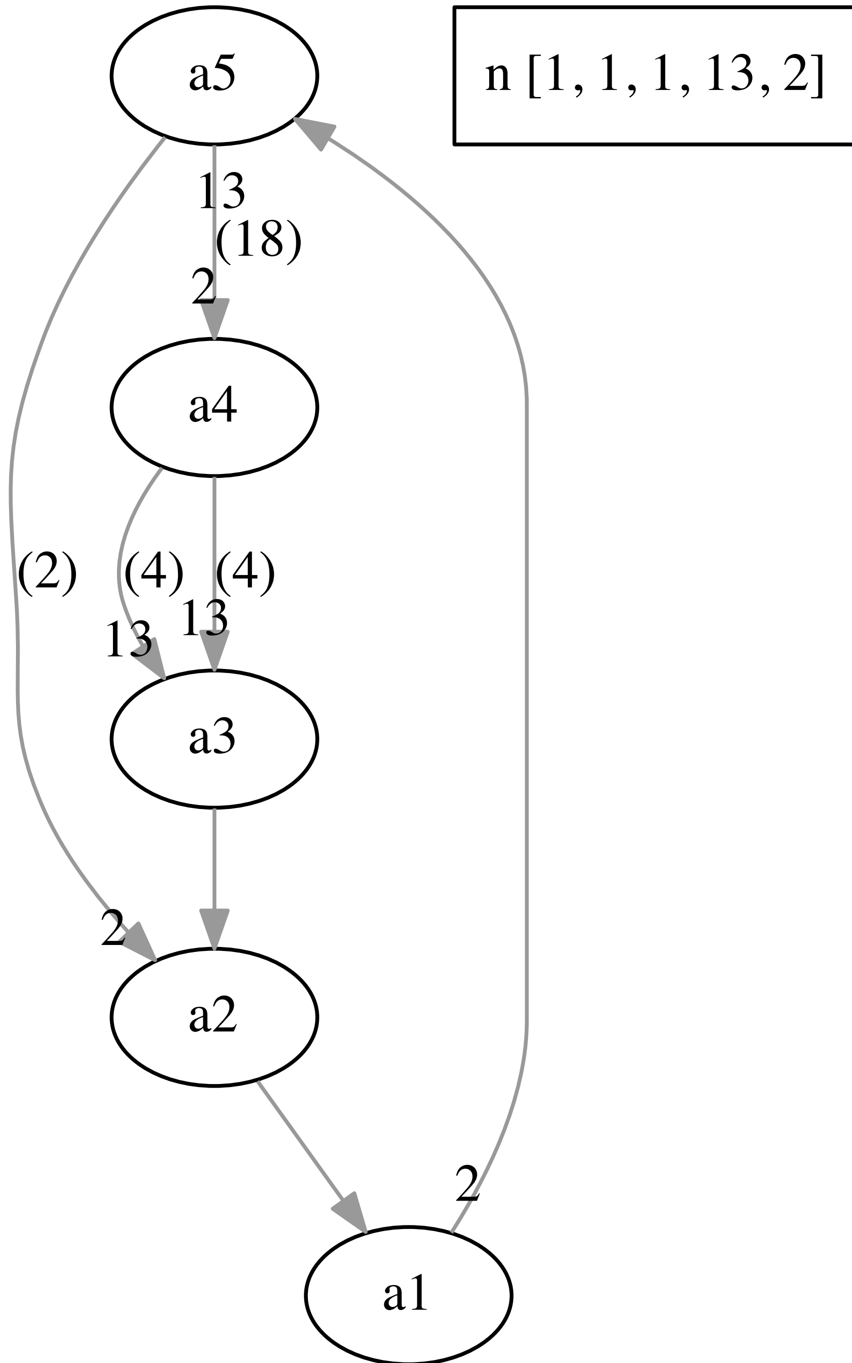


Figure A.12: Artifical instance 7

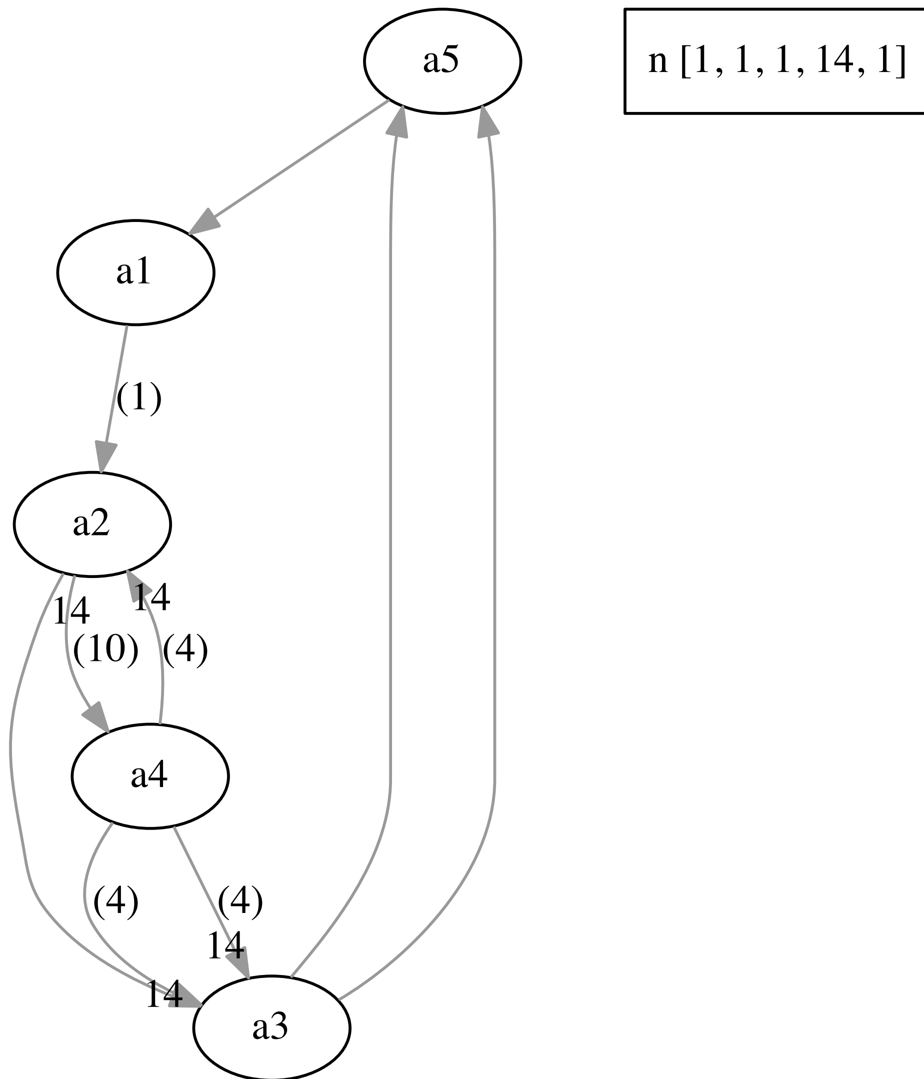


Figure A.13: Artifical instance 8

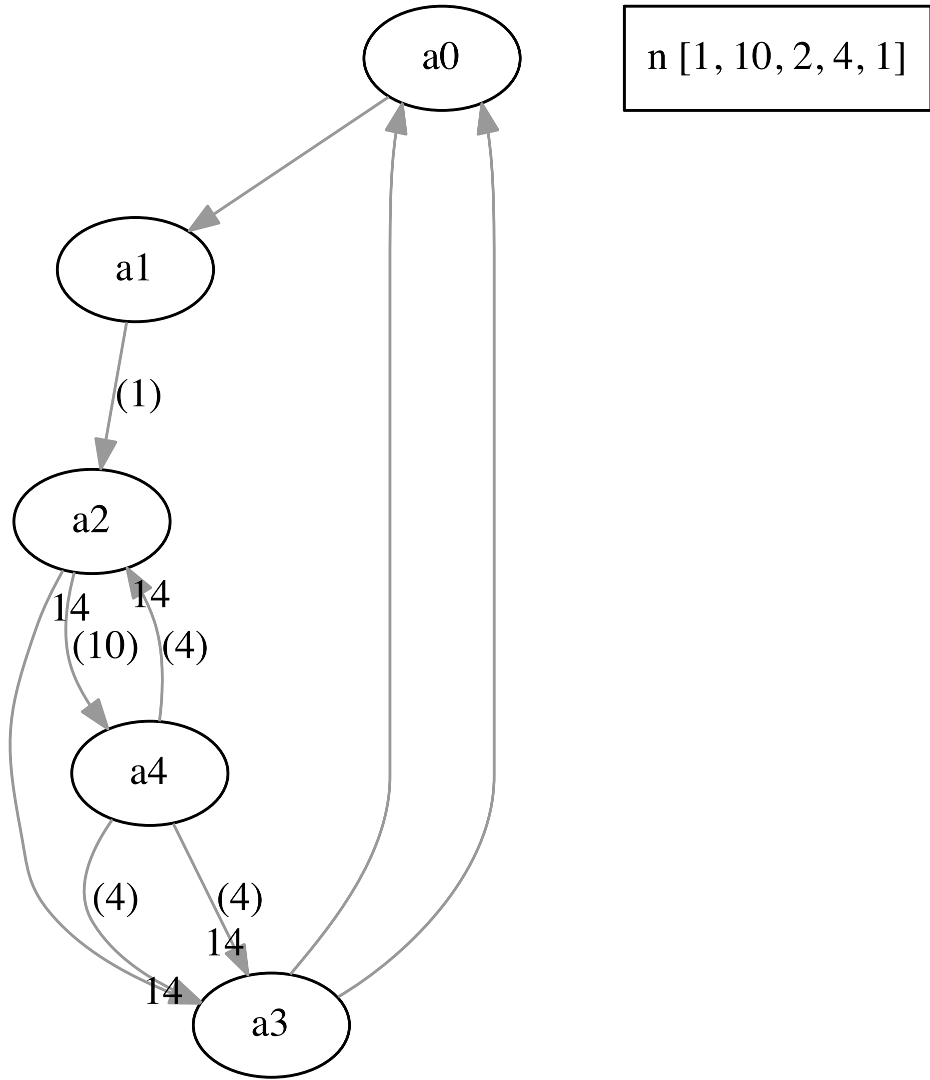


Figure A.14: Artifical instance 9

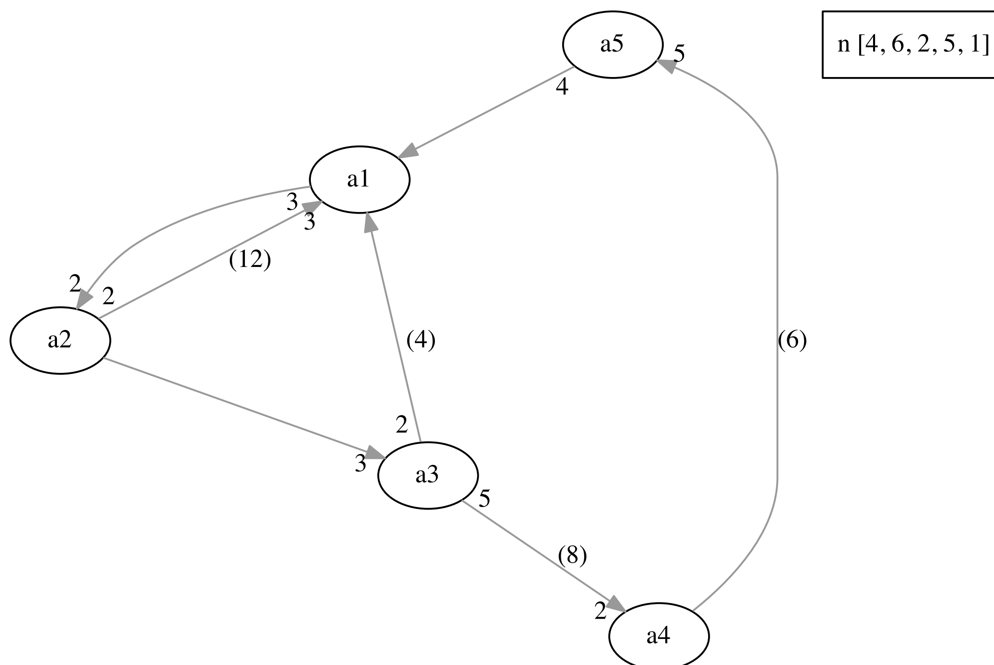


Figure A.15: Artifical instance 10

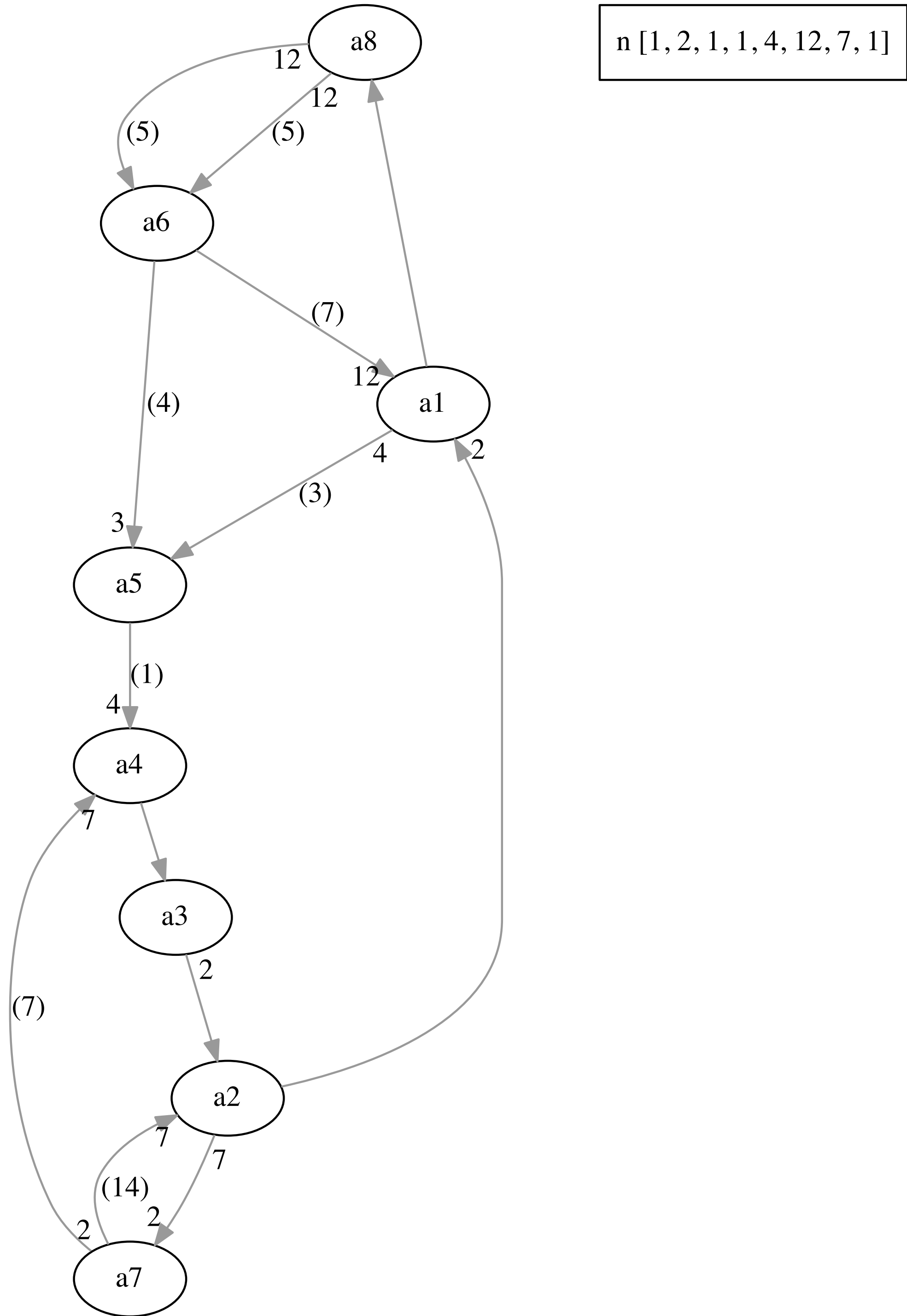


Figure A.16: Artifical instance 11

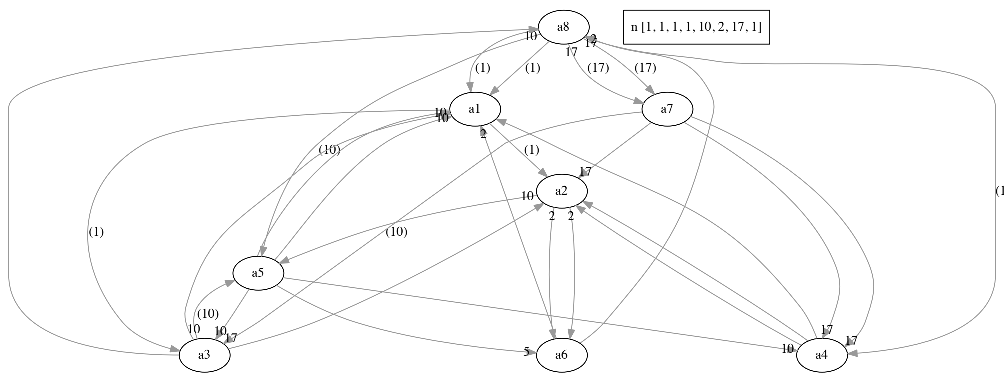


Figure A.17: Artifical instance 12

Appendix B

Nomenclature

API	Application interface
CP	Constraint programming
DAG	Directed acyclic graph
FPS	Frames per second
GPU	Graphics processing unit
HSDF	Homogeneous synchronous dataflow model
ILP	Integer linear programming
MCM	Maximum cycle mean
MPSoC	Multiprocessor system-on-chip
NoC	Network on chip
SDF	Synchronous dataflow
TDMA	Time division multiple access
UAV	Unmanned aerial vehicle
WCET	Worst-case execution time

Appendix C

Content of attached CD

```
.
├── Java\ implementation
│   └── MappingRTStreamingApplicationsOnMPSoC
└── LaTeX
    ├── Chapters
    ├── Hron-thesis-2009.pdf
    ├── Hron-thesis-2009.ps
    ├── Hron-thesis-2009.tex
    ├── Images
    ├── Makefile
    ├── figures
    ├── hyphen.tex
    ├── k336_thesis_macros.sty
    └── reference.bib
```

Figure C.1: Content of attached CD